

The Waite Group


PLUME
WAITE

MASTERING BASIC ON THE TRS-80® MODEL 100

THE COMPREHENSIVE PRIMER FOR
MODEL 100 PORTABLE BASIC

- Features graphics, sound generation, and file handling
- Imaginative, easy-to-follow program examples
- Takes you from novice to advanced programming
- Covers the new, optional disk drive and video interface

by
Bernd Enders



Another book by the bestselling authors of *ASSEMBLY LANGUAGE PRIMER FOR THE IBM® PC & XT* and *BLUE-BOOK OF ASSEMBLY ROUTINES FOR THE IBM® PC & XT*. . . And rave reviews for The Waite Group:

"An outstanding example of how to write a technical book for the beginner . . . refreshingly enjoyable . . . accurate, readable, understandable, and indispensable. Don't stay home without it."

— Ken Barber, reviewing *CP/M Primer*,
in *Microcomputing*

"Mitch Waite . . . has left a distinctive contribution to the literature of computer graphics . . . seeing it here is like understanding it for the first time."

— *Computer Graphics Primer*, reviewed in
Computer Graphics World

"It's hard to imagine that a field only a decade old already has a classic, but Waite's book is just that."

— Tony Dirksen, reviewing *Computer Graphics Primer* in
Interface Age

"... an outstanding reference work, aside from its obvious benefit as an instructional text . . . superb writing style . . ."

— Chuck Dougherty, reviewing *Soul of CP/M* in *Cider*

"The demystification of a complex technological development . . . rating: 100."

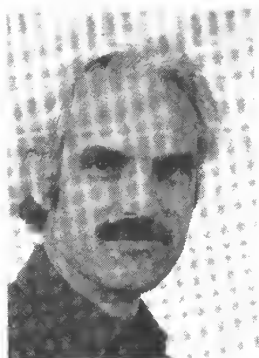
— *8086/8088 16-bit Microprocessor Primer*, reviewed in
The Reader's Guide to Microcomputer Books

"[This book] can have anybody writing and understanding assembly language programs within one hour."

— Alan Neibauer, reviewing *Soul of CP/M*,
in *80 Microcomputing*

"All-inclusive, beautifully organized, easy-to-use reference that helps you wrest order from chaos . . ."

— *CP/M Bible*, reviewed in *Byte Book Club Bulletin*



Bernd Enders

Bernd Enders has been teaching physics and astronomy at the College of Marin for the last thirteen years, and more recently, an introductory computer science course on UNIX. Mr. Enders is also coauthor of the recent book *BASIC Primer for the IBM® PC and XT* (New York: Plume/Waite, New American Library, 1984) and is one of the founders of StarSoft, a small company specializing in astronomical software. He received a B.S. in physics from the University of Chicago and an M.S. in the same field from the University of California, Berkeley. His avocations include woodworking, playing the cello, and painting.

MASTERING BASIC

ON THE TRS-80[®]

MODEL 100

by Bernd Enders



A Plume/Waite Book
New American Library
New York and Scarborough, Ontario

NAL BOOKS ARE AVAILABLE AT QUANTITY DISCOUNTS WHEN USED TO PROMOTE PRODUCTS OR SERVICES. FOR INFORMATION PLEASE WRITE TO PREMIUM MARKETING DIVISION, NEW AMERICAN LIBRARY, 1633 BROADWAY, NEW YORK, NEW YORK 10019.

Copyright © 1984 by The Waite Group, Inc. All rights reserved. For information address New American Library.

Several trademarks and/or service marks appear in this book. The companies listed below are the owners of the trademarks and/or service marks following their names.

Microsoft: MS-DOS, MBASIC
MicroPro International Corporation: WordStar
Tandy Corporation: TRS-80 Model 100 Portable Computer



PLUME TRADEMARK REG. U.S. PAT. OFF. AND FOREIGN COUNTRIES
REGISTERED TRADEMARK — MARCA REGISTRADA
HECHO EN WESTFORD, MASS., U.S.A.

SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN and NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, New York 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L1M8

Cover design by Michael Manwaring
Illustrations by Winston and Karen Sin
Typography by Walker Graphics

First Printing, November, 1984

1 2 3 4 5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

Contents

Acknowledgments	viii
Preface	ix
1 BASIC and Your Model 100	1
What Is BASIC?	1
What You Need to Get Started	2
Starting BASIC	4
2 Your First BASIC Program	7
BASIC's Vocabulary	7
The Direct Mode — Your First BASIC Instructions	9
Your First BASIC Program	11
Summary	16
Exercises	17
3 BASIC Output with PRINT	19
Printing Strings	19
Printing Numbers and Doing a Little Arithmetic	24
Printing with Semicolons and Spaces	26
Printing with Commas	29
Printing with TAB — Controlling Your Columns	31
The PRINT @ Statement — Better Control over Your Screen	32
Summary	36
Exercises	37
4 Housekeeping Chores	39
Special Keys — Making the Programmer's Life Easier	39
Editing in the EDIT Mode	44
LIST and RUN Revisited — Getting More Particular	52
The REM Statement — Keeping Order Within Your Programs	54
Using Your Printer	56
Summary	61
Exercises	62
5 Storing Your Program	64
Storing Your Programs in RAM Files	64
Cassette Files	79
Disk Files	85
Summary	88
Exercises	88

6	<i>Introduction to Variables</i>	91
	Numeric Variables 91	
	String Variables 99	
	Numeric Expressions 103	
	Summary 108	
	Exercises 109	
7	<i>INPUT to Your Program</i>	112
	The BASIC INPUT Statement 113	
	INPUT with Instructions 116	
	INPUT with Multiple Variables 118	
	INPUT of Strings with Commas 120	
	Summary 122	
	Exercises 122	
8	<i>Branching and Decisions</i>	125
	The GOTO Statement — No Two Ways About It 126	
	The FOR...NEXT...STEP Loop 129	
	Letting the Computer Make Decisions 140	
	Summary 147	
	Exercises 147	
9	<i>Dealing with Numbers</i>	150
	Different Kinds of Numbers 151	
	PRINT USING — Printing Numbers a Better Way 162	
	Summary 159	
	Exercises 170	
10	<i>Character Graphics</i>	172
	Of Pixels and Characters 172	
	Direct Entry of Special Characters 173	
	The Extended Character Set with ASCII 179	
	Summary 183	
	Exercises 183	
11	<i>Subroutines and Program Organization</i>	185
	The Subroutine — The Concept 185	
	GOSUB...RETURN and the BASIC Subroutine 186	
	Indexed Branching with ON...GOSUB 191	
	Calling a Subroutine Anytime with KEY() ON and ON...KEY GOSUB 196	
	Summary 202	
	Exercises 202	
12	<i>Data and Arrays — Organizing Information</i>	206
	Storing and Reading Information with DATA and READ 206	
	Arrays, Subscripts, and Dimensions 213	
	Summary 223	
	Exercises 223	

13	<i>Dot Graphics</i>	226
	Putting Points on the Screen with PSET 227	
	Erasing Dots with PRESET 233	
	Drawing Lines and Boxes with LINE 234	
	Summary 241	
	Exercises 241	
14	<i>Numeric Functions</i>	245
	Numeric Functions — An Introduction Using ABS and SGN 246	
	Making Integers with CINT, FIX, and INT 247	
	Trigonometry on the Model 100 251	
	Of Powers, Roots, and Logs 257	
	Random Numbers with RND 261	
	Summary 264	
	Exercises 265	
15	<i>String Functions</i>	267
	The SPACE\$ Function 268	
	The VAL and STR\$ Functions 269	
	Making Strings with STRING\$ 271	
	Input Using INPUT\$ and INKEY\$ 273	
	Manipulating Strings 278	
	Summary 285	
	Exercises 286	
16	<i>Sound and Music</i>	287
	The Basic SOUND Statement 287	
	Playing Music 291	
	Summary 300	
	Exercises 301	
17	<i>Data Files</i>	304
	Storing Information in a Data File 305	
	A Second Look at Data Files 312	
	Summary 322	
	Exercises 323	
	<i>Appendices</i>	325
	A. Reserved Words 325	
	B. ASCII Character Codes 327	
	C. BASIC Error Messages 334	
	<i>Index</i>	336

Acknowledgments

I would like to give special thanks to Robert Lafore, the editor of this book, for his many helpful suggestions and his encouragement throughout this project, and to my friend and colleague Robert Petersen, who has provided invaluable technical assistance and moral support. I would also like to give thanks to my other colleagues at the College of Marin, who had to put up with my single-minded, sometimes obsessive, involvement in writing this book: Don Martin, Steven Prata, Dick Rodgers, and John Hines. And finally, I owe a great debt to Catheryn Zaro for her assistance in preparing the final draft, and her faith that, yes, I would eventually turn off my computer.

Preface

BASIC is one of the easiest computer languages to learn, and one of the most popular in the computer field. It is also the computer language “spoken” by the TRS-80® Model 100 portable computer. This book will teach you how to program in BASIC on your Model 100.

Why Learn BASIC?

The Model 100 is an easy-to-use yet powerful personal computer. Anyone, including those with no previous experience with computers, can very quickly learn to use the Model 100 for such purposes as writing a letter or creating an address list. But suppose you want to use your computer to make financial projections, keep track of your expenses while on a trip, or simply entertain yourself with a game while waiting for a flight to New York? You can buy “canned” commercially available programs to perform some of these functions, but such programs may not operate in exactly the way you want. If you can program in BASIC, you will be able to write powerful programs that are tailored to your individual needs. Learning BASIC is also a stimulating intellectual exercise that will introduce you to the fundamental concepts used in computers and computer programming.

Who Needs This Book?

This book is written for anyone interested in learning BASIC on the Model 100, *especially the person who has never done any programming before*. We assume that you have no prior knowledge of computers or programming; yet, by the time you’ve reached the end of this book, you’ll be able to write some very sophisticated BASIC programs.

How Does This Book Teach BASIC?

The approach used in this book is practical and down-to-earth. Each new BASIC instruction or concept is introduced with examples that you are invited to try out for yourself on the Model 100. To make it as easy as possible to learn BASIC, each chapter opens with a list of the concepts and

BASIC instructions presented in the body of the chapter. As the chapter proceeds, boxed summaries of important statements, commands, and other elements of BASIC recap the material you have just read. Finally, each chapter ends with a summary that provides an overview of the chapter's contents, followed by a selection of exercises, with solutions, that will help you check your understanding of what you have learned.

We think you'll find this book informative, enjoyable, and exceptionally easy to read — whether you're a computer novice or an experienced programmer for whom BASIC is a second or even a third computer language. Enjoy yourself!*

*Some of the material in this book originally appeared in a different form in *BASIC Primer for the IBM® PC and XT*, by Bernd Enders and Bob Petersen (New York: Plume/Waite, New American Library, 1984).

1

BASIC and Your Model 100

Concepts

BASIC as a computer language

What you need to get started

The main menu and selecting BASIC

Welcome to the world of BASIC. In this chapter we provide the background you need to begin learning how to write BASIC programs on the TRS-80® Model 100. We'll explain what BASIC is and what some of its special features are. We'll also discuss some of the optional devices you can attach to your Model 100. Finally, at the end of the chapter, we'll explain how to get ready to write your first BASIC program.

What Is BASIC?

A computer is really nothing more than a large collection of switches — and we do mean *large* — millions of them! These switches are like the light switches in your home in that they are either in an on or off position; this is how information is stored in a computer. All the incredible things that computers do, such as making complex financial projections and piloting the Space Shuttle, are done by manipulating these millions of switches. Most of us certainly don't want to communicate with computers at that level of ons and offs — that's a difficult, specialized, and time-consuming job. As nonspecialists, we want to be able to “talk” to a computer in a language that is as close as possible to our own way of thinking and communicating. Your Model 100 doesn't understand English, but it does understand BASIC, which is perhaps closer to English than any other computer language. BASIC is a compromise between the natural on-off language of the machine and our own language, English.

In addition to being quite a bit like English, BASIC is easy to learn because it provides immediate feedback — many computer languages do not. If you're not sure about something, you can simply try it and see if it works! BASIC will let you know right away if something is wrong by printing out what is called an "error message".

As convenient as it is, BASIC is also a powerful and versatile language that you can use to write programs in many different fields. For example, typical business-related programs might keep track of your finances, make financial projections, analyze your investment portfolio, or generate mailing labels. Or you can write BASIC programs to keep track of your diet, balance your checkbook, or entertain yourself, family, and friends with a clever computer game you invented.

BASIC on your Model 100 can, of course, also be used for mathematical and scientific applications, ranging from such topics as modeling the motion of a bouncing ball to simulating the motions of the planets. Problems that once required literally years of hand calculations can be done on your Model 100 in a matter of seconds.

BASIC on your Model 100 has several graphics-related enhancements that allow you to draw various special characters, points, and lines with single BASIC instructions. These capabilities make it easy, for example, to graph the performance of your favorite stock. The graphics on your Model 100 are also well suited to computer games. A great variety of special graphics characters are available that can be moved about the screen with surprising speed.

What You Need to Get Started

If you have a TRS-80 Model 100 portable computer, you're ready to start learning BASIC with this book. Even the most inexpensive model of the Model 100 comes with everything you'll need to learn BASIC. There are, however, some optional devices that add computing power to your system and may make certain tasks easier. Let's discuss these optional additions one at a time.

Additional Memory

The most inexpensive Model 100 has eight *kilobytes* (or *K-bytes*) of memory available to the user. We needn't go into a detailed discussion about computer memory here, but it is helpful to know that the number of bytes refers to the amount of information you can store in your computer. Memory in a computer is similar to the amount of space you have in a filing cabinet. Eight kilobytes (equal to 8,192 bytes) is enough memory to store

the equivalent of about 2,000 words. That's more than enough memory for any of the BASIC programs in this book.

However, if you want to store many BASIC programs at the same time, or if you want to store information created by some of the built-in application programs (such as ADDRSS or SCHEDL) in addition to writing BASIC programs, you may find it convenient, even necessary, to add more memory. Additional memory is available in eight-kilobyte increments. The maximum amount of memory you can have in your Model 100 is 32 kilobytes.

A Printer

A printer is another valuable addition to your Model 100, but again, it's not essential. A printer gives you permanent records of your programs and their output. Also, when writing long programs, it is helpful to be able to see the whole program at a glance in order to locate errors and avoid getting lost. Your screen can show only eight program lines, but a printer can print out the whole program, no matter how long it is.

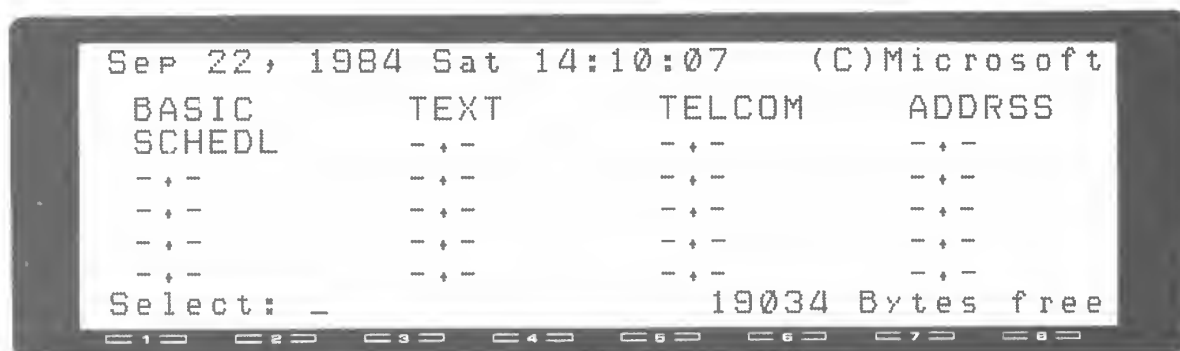
A Cassette Recorder

One outstanding feature of the Model 100 is that it can save up to nineteen different *user files* without the support of the additional storage devices required by other computers. A user file is a body of text or a BASIC program that is stored in your computer's memory under a given filename; if you think of the user file as the information inside a file folder, then your Model 100 has room to store information inside nineteen separate file folders. These user files can be stored permanently in your Model 100 — that is, as long as the batteries are healthy!

The capability of your Model 100 to store up to nineteen different BASIC programs is more than adequate for the purpose of learning BASIC with this book. You can, however, add an additional electronic filing cabinet by attaching a *cassette recorder/player* to your Model 100. There are two advantages to this additional storage device: (1) magnetic tape doesn't rely on batteries to keep the stored information intact, which means that you'll never lose a file because your batteries have gone bad, and (2) if you use your Model 100 for purposes other than learning BASIC, you may find that nineteen user files just aren't enough. A cassette recorder will solve this overflow problem: you can save on cassette tape those files that you use only infrequently.

Starting BASIC

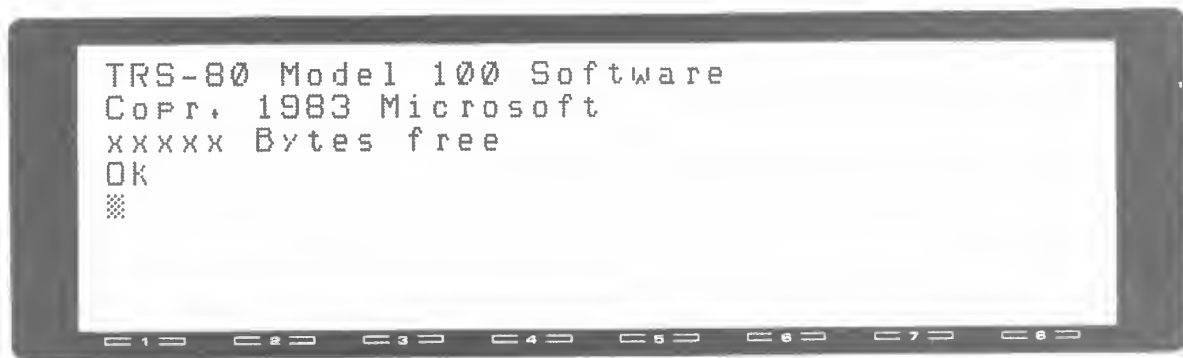
The style of this book is action oriented — we don't like to *talk* too much without actually *doing* something. So before we leave this introductory chapter, let's turn on your Model 100 and prepare it to accept your first BASIC instructions. First, turn on your Model 100 by means of the on-off switch on the right side. The display below, called the *main menu*, should immediately appear on your screen. If you have difficulty reading it, adjust the Display Adjustment Dial next to the on-off switch for optimum readability or tilt the machine slightly to change your viewing angle.



Your screen now shows the date and time and lists five built-in *application programs*: BASIC, TEXT, TELCOM, ADDRSS, and SCHEDL. For a complete discussion of all the applications programs, see the book in this series called *Introducing the TRS-80® Model 100* by Diane Burns and S. Venit (New York: Plume/Waite, New American Library, 1984). This book will be concerned mainly with the application program BASIC. In addition to the listed application programs, your screen may also show a number of other files that had been stored previously by a user.

Calling up BASIC in your Model 100 is a simple two-step process. First, make sure the *menu cursor* is on the word *BASIC*. The menu cursor is a shaded rectangle that can be moved around the screen by means of the four cursor control keys at the upper right-hand corner of your keyboard. Underneath these keys are arrows that specify the direction of the cursor movement. If you haven't yet pressed any keys since you turned on your computer, you're all set — the menu cursor is already right on top of the word *BASIC*. Second, locate the **ENTER** key; it's the large key on the right side of your keyboard, as shown in Figure 1-1. Its size reflects its importance — we'll discuss it further in the next chapter.

Now press the **ENTER** key. The menu is immediately replaced by the following information:



The first two lines on the screen tell you that you are using “software” for your Model 100 written by Microsoft. The word *software* refers to all the stored information that allows your computer to carry out all your instructions. The sequence of x’s in the third line is actually a *number* that specifies

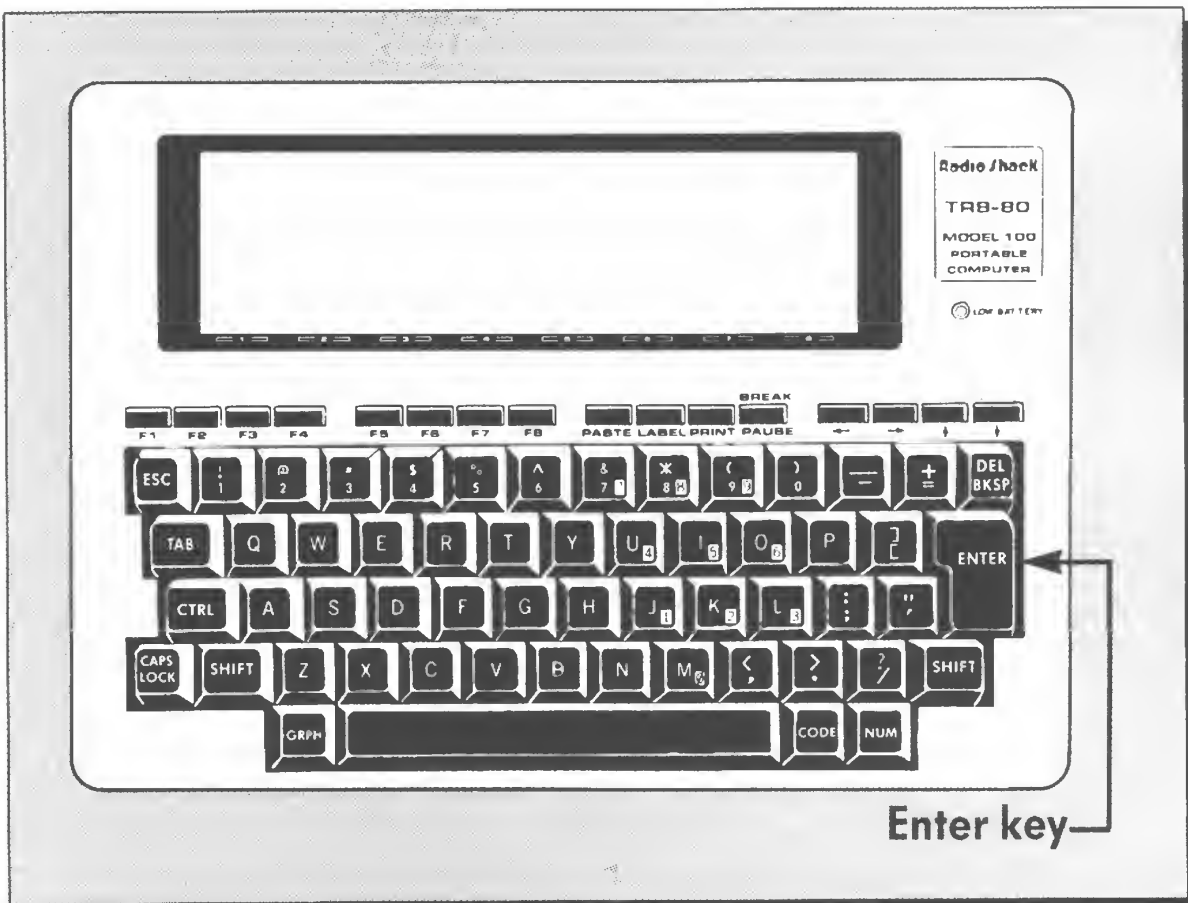


Figure 1-1. The keyboard

how much computer memory is available for your use. The size of this number equals the amount of memory installed in your particular computer minus the amount used up by previously created files.

The most important word on the above screen is the word *Ok*. Called the BASIC “prompt”, it signifies that your computer is in the BASIC Command Mode; that is, your computer is now ready to accept your BASIC instructions.

The blinking square right underneath the “Ok” is the BASIC cursor, which we’ll just call the “cursor” from now on. The cursor is simply a pointer that shows where a character will appear on the screen if you press a key on your keyboard. We’ll see how the cursor works in the next chapter.

You’re ready to try your first BASIC instructions! See you in Chapter 2!

2

Your First BASIC Program

Concepts

- The **ENTER** key
- BASIC's vocabulary
- Syntax errors
- Indirect and Direct Mode
- Simple editing with the **DEL/BKSP** key
- Commands and statements
- Line numbers and programs

Instructions

PRINT, BEEP, LIST, RUN, CLS, END, NEW

*I*n this chapter we will give you your first experience in writing and running a simple BASIC program. Let's assume that you've turned on your Model 100 and have selected BASIC from your screen menu, as described in Chapter 1. This means that you should find the BASIC prompt — an “Ok” — and the blinking cursor, right underneath some introductory information (see Chapter 1).

BASIC's Vocabulary

To get started, let's use the “try it and see” method. Type the phrase “hello, computer”. Each time you press a key, the corresponding letter appears on your screen in place of the cursor, while the cursor moves one space to the right. The cursor shows you *where* whatever you are typing will appear on the screen. Incidentally, if a number appears on your screen when you type a certain letter, such as *k*, turn off the *numeric lock* by pressing the **NUM** key. The letters *NUM* stand for “NUMber”; when this key is in

its depressed position (**NUM** is on), the right side of the Model 100 keyboard behaves as a numeric keypad, a handy feature if you're typing a lot of numbers. Also, if capital letters appear on the screen as you're typing lowercase letters, hit the **CAPS LOCK** key to turn off the *capital letter lock*.

Assuming everything is working as expected, the typed "hello, computer" should appear right under the "Ok", as shown below:

```
OK
hello, computer
```

But this is all you'll see; nothing more happens — until you press the **ENTER** key. We used the **ENTER** key in Chapter 1 to select BASIC from the main menu — it's the large key on the right side of your keyboard. Pressing the **ENTER** key causes your computer to respond with the rather cryptic message "?SN Error", as shown below:

```
OK
hello, computer
?SN Error
OK

```

What does this mean? Well, several things. First, although some sort of error evidently occurred, you didn't break anything! You know everything is all right because the computer says "Ok".

Second, notice that it is not until you press the **ENTER** key that the computer really "reads" and responds to what you typed on the screen. This procedure — typing something and then pressing **ENTER** — is called *entering*. From now on, when we say *enter* "something", we mean type "something" and press **ENTER**.

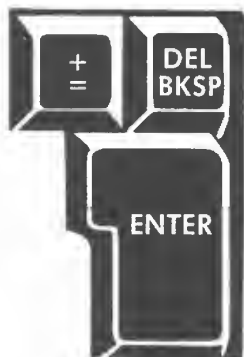
Third, your Model 100 evidently didn't understand what you entered. "?SN Error" is your computer's way of saying, "Huh? I don't understand you". The letters *SN* are an abbreviation for "SyNtax", which refers to the structure or grammar of a word or phrase. The message "Syntax Error" tells you that you've entered a pattern of letters or words that your computer cannot understand. Because people are still smarter than computers, we must make the extra effort to communicate with them — in this case, via the computer language BASIC.

Fortunately, whereas most human languages have a vocabulary of many tens of thousands of words, BASIC on your Model 100 has approximately 150 words. These are called *Reserved Words* and are listed in Appendix A. These BASIC words, either alone or in appropriate combinations, make up the BASIC instructions that your Model 100 can understand. Learning

BASIC is like learning a foreign language in that both have a vocabulary and rules for combining words into phrases. BASIC, however, is much easier to learn than a typical foreign language because its vocabulary is much more restricted and its “grammar” is much simpler.

The **DEL/BKSP** Key — Simple Deletions

Before we get much further into BASIC, we should show you how to correct typing errors (a rather crucial concern to some of us!). Locate the key just above the **ENTER** key. It looks like this:



The letters *DEL* and *BKSP* are abbreviations for “DELeTe” and “BaCK-SPaCe”. That’s what this key does: it causes the cursor to backspace and in the process erase the character that initially appeared to the left of the cursor. To make a correction, type the obviously misspelled phrase “hello, computer”, and then press **DEL/BKSP** three times. The result is “hello, comput”. To complete the correction, type the letters *er* to give the corrected phrase “hello, computer”.

The Direct Mode — Your First BASIC Instructions

Now let’s enter something your computer can understand. Instead of entering the phrase “hello, computer” as we did previously, enter (remember, *enter* means type “something” and then press **ENTER**) the following:

```
Print "hello, computer"
```

This time your computer’s response is more encouraging. The new lines on your screen now look like this:

```
Print "hello, computer"
hello, computer
Ok
⌘
```

← BASIC statement you enter
← Response of your Model 100

As you can see, your computer prints out the phrase inside the quotation marks as soon as you press the **ENTER** key. The word *print* is a legal BASIC word (it's on the Reserved Words list) that causes the subsequent phrase in quotation marks to be printed out. We'll say much more about this very important BASIC instruction in the next chapter.

This use of a BASIC instruction that causes your computer to *execute* (perform whatever task you ask it to perform) immediately after you press **ENTER** is called the *Direct Mode*, and the line entered in this manner is sometimes called an *immediate line*. (In the next section we'll see how this Direct Mode and immediate lines differ from the Indirect Mode and program lines.)

Let's try another BASIC instruction in the Direct Mode. It's one the simplest BASIC commands: just type the word *beep* and press **ENTER**. The new lines on your screen look like this:

```
beep
OK

```

Nothing very exciting happens on the screen, but your computer does what you asked it to do — it beeps as soon as you press the **ENTER** key! In future programs you'll find the BEEP instruction very handy as the computer's way of getting your attention.

To summarize, BASIC instructions in the Direct Mode are executed immediately after the **ENTER** key is pressed.

Your Screen and the CLS Command

If you've tried all the suggested examples, your screen by now will be completely filled up. Because the screen only displays eight lines of text, including the prompt and cursor, it doesn't take much *code* ("computer instructions") to fill it up. What happens when the screen gets full? When your screen is full, the BASIC prompt is at the bottom of the screen. If you now type in a new instruction, such as BEEP, it will appear on the bottom of the screen, the same line on which the cursor appears. Pressing **ENTER** will cause the contents of your screen to *scroll* upward, making room for a new "Ok" at the bottom of the screen, and hence a new entry. Try it a few times by entering "beep" or just pressing the **ENTER** key. Your Model 100 will always make room at the bottom of the screen, so you'll never have to worry about having enough space or about losing your cursor below the

bottom of the screen. Nevertheless, you may prefer to work on an uncluttered screen. If you'd like to erase or "clear" the screen, BASIC's CLS command does that job. Try it by entering "cls":

```
cls
```

The effect of CLS is to clear the screen and to move the "Ok" prompt and the cursor to the *home position*, which is the upper left-hand corner of your screen. You can use the CLS command whenever you want to work on an uncluttered screen.

Your First BASIC Program

Using a BASIC statement in the Direct Mode to perform a single task — as illustrated in the previous examples — has an obvious limitation: you can do only one thing at a time. What if you want the computer to perform many tasks in sequence? Virtually any problem worth buying a computer for involves doing a *series* of BASIC instructions rather than one at a time. What we need is a BASIC *program* — a sequence of BASIC instructions that are stored in the computer's memory and then executed whenever we wish. A BASIC program can consist of just one instruction or of many hundreds. In contrast to the *Direct Mode* commands we have used so far, *programs* operate, by definition, in the *Indirect* or *Program Mode*.

To illustrate how a BASIC program works, let's write a very simple program using the PRINT and BEEP statements you are already familiar with. Clear the screen and type the following line:

```
10 print "My first BASIC program!"
```

Don't forget the number 10 before "print" — this number is very important, as you'll see shortly. Now press the **ENTER** key. You'll notice that nothing happens — the computer doesn't print the phrase in quotes, as it did in Direct Mode. Now type the second line:

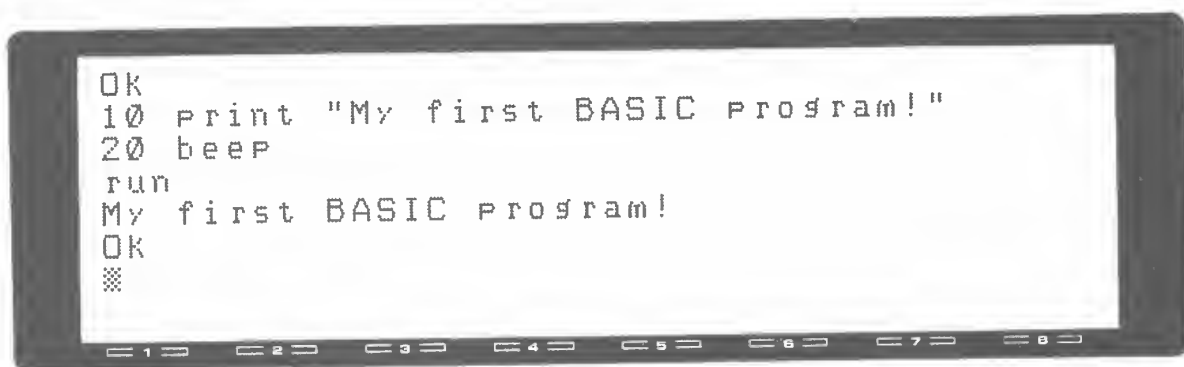
```
20 beep
```

Remember to press **ENTER**. The whole program should now look like this:

```
10 print "My first BASIC program!"  
20 beep
```

Although typing these lines may not seem very effective (because the computer didn't do anything), something *did* happen in the innards of your computer's memory: every time you pressed **ENTER**, the computer read and put into its memory the BASIC instructions you typed on that line. To make the computer actually *execute* this sequence of BASIC statements, you need only enter the command RUN.

Enter the word *run* now. The result is not totally unexpected: first your computer prints "My first BASIC program!" on your screen, and then it beeps. Your screen now looks like this:



Line Numbers and Program Lines

Let's look at some details of the program we just constructed. Undoubtedly, you will have wondered about the numbers that begin each line. These are called *line numbers* and they have two very important functions.

First, line numbers tell the computer that the statements you have entered are part of a *program* rather than just a series of commands to be executed the instant you press **ENTER**. As you can see, a program is simply a sequence of numbered BASIC instructions, called *program lines*, that are executed whenever you enter RUN. This is quite different from how we first used BASIC instructions in the Direct Mode, in which BASIC commands are carried out immediately after **ENTER** is pressed. We will refer to BASIC instructions used within program lines — that is, instructions preceded by line numbers — as *BASIC statements*. We'll call instructions used in the Direct Mode — that is, without line numbers — *BASIC commands*.

The second function of line numbers is to define the *order* in which the computer carries out your BASIC statements. In a simple program like the one we just wrote, execution always proceeds from the smallest to the largest line number; thus line 10 is executed before line 20. But why did we skip all the numbers between 10 and 20? The answer is that we *could* have used 1, 2, and 3, and the program would have worked just as well. However, leaving room between line numbers gives you the option of inserting addi-

tional BASIC statements between existing numbers later on, in case you want to modify or add to your program. Suppose, for example, that you want to add a beep before your computer prints out “My first BASIC program!”. This kind of change is easily made in BASIC. Just enter the following new line:

```
5 beep
```

Now run your program again by entering RUN. Your Model 100 does just what it is supposed to do: it beeps, then writes “My first BASIC program!” onto the screen, and then beeps again.

To summarize, we have inserted a new BASIC statement into an existing program simply by entering a new line with an appropriate line number. Writing programs with line numbers that are multiples of 10 is a common and sensible practice, because it usually assures you of plenty of room to insert new statements.

Correcting and Deleting Program Lines

In addition to inserting lines, you can also correct or delete lines you have already entered. In order to change a line, you need only reenter the same line and line number with the desired changes. (Another way to make corrections is to use the built-in BASIC editor, which we’ll discuss in Chapter 4.)

For example, if you want to change line 10 to say “My first program works!” and to delete the second beep, you would enter the following:

```
10 print "My first program works!"  
20      ← That's right, just type '20' and enter it  
OK
```

Now enter “run” — sure enough, you hear a beep, and then “My first program works!” appears on the screen. Evidently, the computer simply substituted the new contents of lines 10 and 20 for the old contents. The new line 20 with nothing following it has the effect of simply erasing whatever was there before; so an easy way to delete a BASIC line is simply to enter the line number of the line to be deleted.

The Two Functions of Line Numbers

1. Line numbers define program lines, which contain BASIC instructions that are stored as part of a program for later execution (in contrast to commands or immediate lines, which are executed immediately).
2. Line numbers define the order in which BASIC statements will be executed when the program is run.

LIST — Displaying Your Program

With all the changes we have made in our original program, it's easy to lose track of what our present program actually looks like. An easy way to find out is to enter "list". The LIST command does exactly what it suggests: it lists the current contents of your program.

Before we try this new command, however, let's clear the screen by entering "cls". You needn't worry that CLS will erase your program; it only clears the screen. The program remains safe and sound in the computer's memory.

Now with the screen cleared, enter "list". Your screen should look like this:



This is your current program. Notice, however, that it is displayed somewhat differently than you entered it: the words *BEEP* and *PRINT* are capitalized, even though you may have entered them in lowercase. LIST always capitalizes words that are part of the BASIC vocabulary, regardless of how you entered them. The advantage of this feature is that your program is listed more clearly, while you are spared the inconvenience of having to use the

SHIFT key. From now on, we will follow suit and write all BASIC words in capital letters; however, you may enter them as lowercase letters.

Another helpful feature of LIST is that it rearranges your program lines in proper sequence. Notice that line 5 has been properly inserted before line 10.

Use the LIST command whenever you want to look at your program, especially if you feel that you are beginning to lose track of what your program looks like.

END — Knowing When to Stop

Your previous program, as listed above, works perfectly well as it stands. When you enter RUN, BASIC starts execution with the lowest line number and finishes with the largest number. When you enter RUN again, BASIC knows that it should start at the beginning, that is, with the lowest line number.

In more complex programs, however, there may well be program lines *after* the point at which you want the program to end (an example is the *subroutine*, which will be explained in a later chapter). The important point here is that a subroutine can have larger line numbers than the line number of the last statement to be executed. In such a case, it is crucial to define the end of a program with an END statement. END is a BASIC instruction that causes the computer to stop execution and to return to the BASIC command level (signified by the “Ok” prompt). In order to develop good habits and avoid unnecessary problems in the future, it is a good idea to end all programs with the END statement. To illustrate, let’s add an END statement to our current program. Enter the line

```
20 END
```

and then enter LIST to see if the computer was really listening. This is what should appear on the screen:

```
LIST
5 BEEP
10 PRINT "My first program works!"
20 END
OK
⌘
```

To be sure that **END** really works as described above, add a line after the **END** statement, like

```
30 PRINT "this statement is past END"
```

If you now run this program, you should get exactly the same output as before. The computer never gets to line 30 because **END** terminates execution at line 20.

NEW — Erasing the Current Program

You have now successfully completed writing, modifying, and running your first BASIC program, and you may be inclined to try some new programs of your own. To begin a new program, it would be convenient to *erase* the program that you wrote before, like erasing a blackboard before you start writing again.

You might be tempted to erase your current program by simply turning your Model 100 off and back on — go ahead and try it! Remember that when the menu appears you must press **ENTER** to get back into BASIC. Then enter **LIST** to see what your Model 100 remembers. Lo and behold, your program is still there! Your Model 100 didn't "forget" the current program, even though you turned it off. Most computers do forget when you turn them off, but your Model 100 has a special kind of memory that functions as long as the batteries do.

So how can we erase the current program? We do it with the **NEW** command. **NEW** simply erases the current program — the program that appears when you enter **LIST** — and readies the computer to receive new instructions. Try this command. Enter **NEW**:

```
NEW
```

Now enter **LIST** to see whether the program is gone. Nothing is listed; the old program has indeed been erased.

Summary

You now know some of the rudiments of BASIC programming. You've learned that a BASIC command (or immediate line) is a BASIC instruction that is executed immediately after you press the **ENTER** key. A BASIC statement (or program line), on the other hand, is a BASIC instruction that, when entered, is "remembered" but not immediately executed by the computer. BASIC statements begin with line numbers, while BASIC commands

do not. BASIC statements are executed only after RUN is entered. A BASIC program comprises a series of program lines that allow the computer to carry out a potentially large set of instructions whenever RUN is entered.

Here is a list of BASIC instructions introduced in this chapter, along with their effects:

PRINT "hello"	Causes your Model 100 to PRINT what is inside the quotation marks.
BEEP	Causes your Model 100 to "beep."
CLS	Clears the screen, but not the current program.
LIST	Causes your computer to "list" the current program.
RUN	Causes your computer to execute the current program.
END	Terminates program execution.
NEW	Erases the current program in the computer's memory.



Exercises

1. Clear the current program and then write, LIST, and RUN a program that will do the following: (1) print out "My computer can't talk" on the first line; (2) print out "but it can chirp like a bird" on the second line; and (3) make a beep sound. Try writing this program before you look at our solution.

2. Modify the program written in exercise 1 so that the *program* clears the screen before it PRINTs and BEEPs. Hint: BASIC instructions that can be used as direct commands (like CLS) can (usually) also be used within program lines as BASIC statements.

Solutions

1. Enter NEW and the program lines shown below:

```
10 print "My computer can't talk"
20 print "but it can chirp like a bird"
30 beep
40 end
```

After you clear the screen with CLS and list the current program, your screen should display the following:



Clear the screen again and enter RUN. The following is the output:

```
OK
RUN
My computer can't talk
but it can chirp like a bird
OK
⋈
```

← Your Model 100 now 'beeps'

2. Enter the following new line:

```
5 CLS
```

Now LIST and RUN your modified program to check the results.

3

BASIC Output with PRINT

Concepts

- Printing strings and numbers
- Length of logical BASIC line
- Centering a title
- Printing columns
- Arithmetic operations
- Character coordinates
- Use of semicolons and commas

Instructions

PRINT, TAB, PRINT @

One important aspect of programming is the capacity to generate output to your screen in a useful and appealing format. In this chapter we will explain in some detail the powerful PRINT statement, which can be used to print both text and the results of simple arithmetic operations. To enhance your ability to print material anywhere on the screen, we will also introduce the BASIC instruction TAB and the PRINT @ statement, an extension of PRINT.

Printing Strings

In Chapter 2 you became familiar with one use of the PRINT statement — printing any sequence of characters enclosed by quotation marks. For example, the program

```
10 PRINT "BASIC is easy"  
20 PRINT "*****"  
30 PRINT "$2.50"  
40 END
```

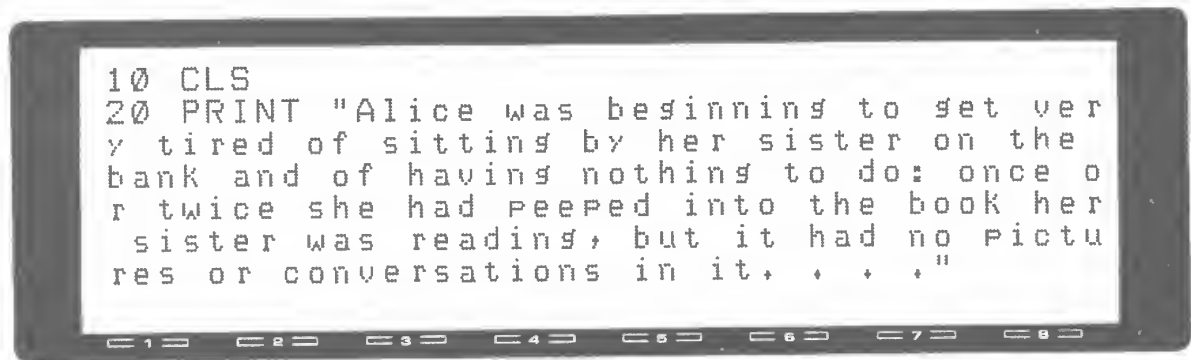
will result in the following screen output when you enter RUN:

```
RUN
BASIC is easy
*****
$2.50
OK
```

As you can see, the sequence of characters enclosed by the quotation marks is printed out exactly as written in the PRINT statement. A character can be anything you can find on the typewriter part of your keyboard, including numbers, spaces, dollar signs, and asterisks. A sequence of characters bracketed by double quotation marks is called a *string*.

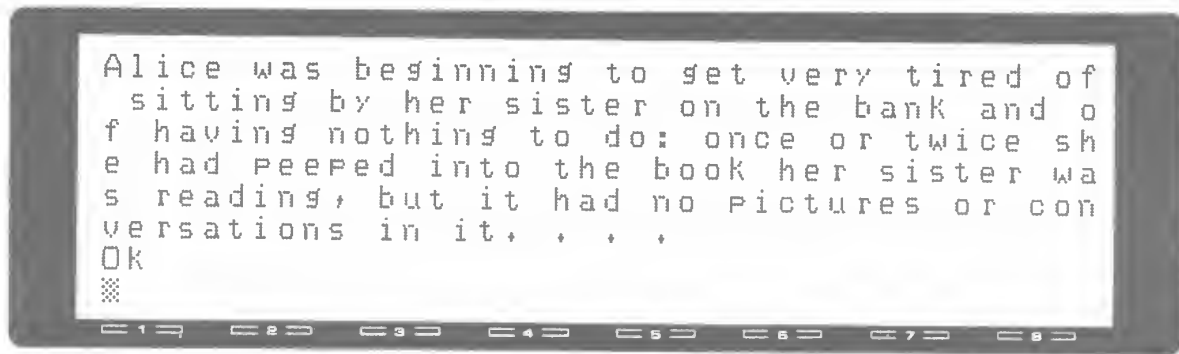
Dealing with Long Strings and BASIC Lines

One question we must answer sooner or later is “How long can a string be?” One way we can find out is to simply try PRINTing a very long string, such as the following:



As you're typing this rather long string, notice how the text automatically wraps around to start at the next line whenever you get to the right boundary of your screen. The **ENTER** key should be pressed only after you've finished typing the whole string — that is, when you've finished the whole BASIC statement, sometimes called the *logical* BASIC line (in contrast to the actual *physical* or *screen* line). In our example of line 20, the logical line is 231 characters long, whereas each screen line is, of course, only 40 characters long — the width of the Model 100 screen.

RUNning this program results in the following output:



There! No error message, just the opening sentences of *Alice in Wonderland* that we asked BASIC to PRINT. The string has 220 characters, and line 20, which PRINTed this string, has 231 characters. How long can we make a string, and how long can a BASIC logical line be? The answer to both questions is 255 characters. That's *long* — long enough to almost fill up your entire Model 100 screen and long enough that, for most applications, you needn't worry about exceeding either the maximum string length or the maximum length of a BASIC line.

Centering a Title — An Example Using Strings

To illustrate the usefulness of strings, let's write a program to print the centered and underlined title "HOW TO BECOME RICH". On a typewriter, we produce an underlined phrase by typing the phrase and then going back over it with the underline key. Unfortunately, we can't underline in the same way on the computer screen because we can't write two characters on top of each other (in this case, a letter and the underline character). Our previous example, however, suggests another way to do it: just use the dash (or the asterisk) on the line *below* the title.

In order to center our title, we need only add the correct number of spaces in front of the title (but after the quotation marks) so that the title appears centered. We can do this by trial and error (always acceptable!) or by doing a little arithmetic to find the number of spaces we need: just subtract the number of characters in the title from the number of characters that fit into a whole line (often called the *screen width*) and divide by 2. We count 18 characters in the title (including spaces, of course), and the screen width is 40 characters: $40 - 18$ equals 22, and 22 divided by 2 equals 11.

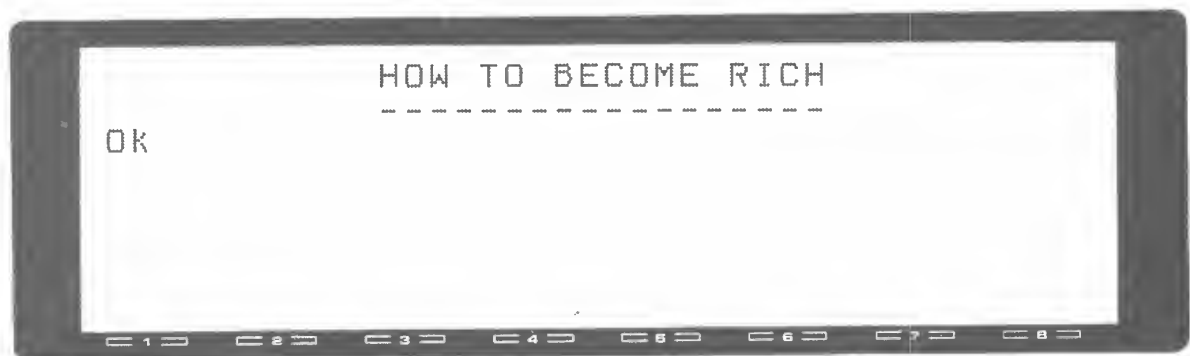
So we need to insert 11 spaces between the title and the left edge of the screen. The following program should do the trick:



```
OK
10 PRINT "                HOW TO BECOME RICH"
20 PRINT "                -----"
30 END
OK
```

The blank lines between lines 10 and 20 and between 20 and 30 result from the wraparound phenomenon we discussed earlier. Here are the details of how this happens. The last quotation mark in line 10 happens to be in column 40, the last character position on the screen line. When you type this quotation mark, the cursor automatically moves to the next character position, which is on the *next* line. Hence line 10 takes up *two* screen lines. Pressing **ENTER** results in the usual carriage return that moves the cursor down one *more* line to the position where you can now enter line 20. Similarly, line 20 also takes up two lines.

Clearing the screen and running this program results in the following:



```
OK
                HOW TO BECOME RICH
                -----
```

Vertical Formatting

Let's take this example a bit further. Can we print a title (or any text) not at the top of the screen but, say, on the third line from the top? Yes, but we need to learn how to print blank lines.

In the programs we have written so far, you may have noticed that two consecutive PRINT statements have always resulted in two printed lines,

one above the other. This means, in effect, that the first PRINT statement ends with what on a typewriter would be a carriage return. This suggests the possibility of printing blank lines by means of a PRINT statement with nothing following it. With this idea in mind, let's modify our existing program by entering the following new lines:

```
2 CLS
4 PRINT
6 PRINT
```

Because the line numbers of the statements are less than 10, they should insert themselves before line 10 of our original program.

Let's enter LIST to check that everything is as it should be:

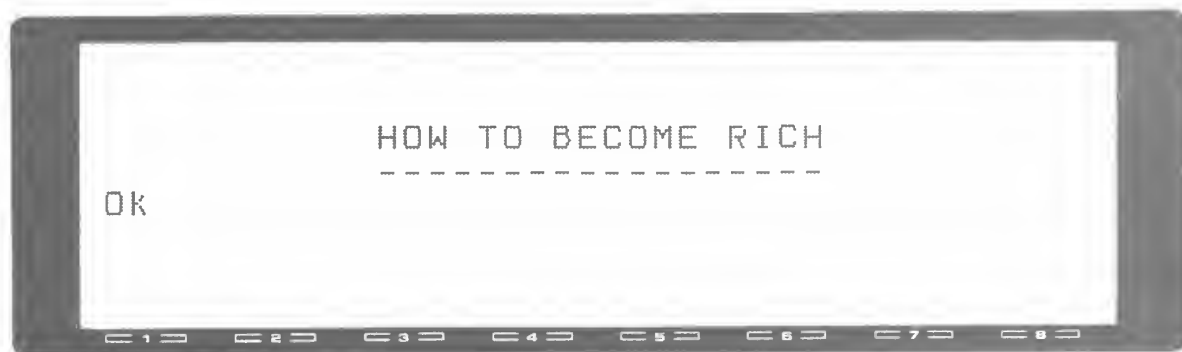
```
LIST
2 CLS
4 PRINT
6 PRINT
10 PRINT "          HOW TO BECOME RICH"

20 PRINT "          -----"

30 END
OK
```

Line 2 instructs the computer to clear the screen. Lines 4 and 6 say “print nothing but do a carriage return”. Then lines 10, 20, and 30 print the actual title and underline it. Note the usefulness of having plenty of room between program line numbers here: it makes it easy to insert new lines.

The output of our program looks like this:



Voila — just as we expected! Now we have a program that prints out a title in a rather nice and useful format (to say nothing of the title itself!). In the

process we have learned how to put a phrase anywhere on the screen by the appropriate use of spaces within a string and by the “empty” PRINT statement — that is, a PRINT statement without anything following it.

Printing Numbers and Doing a Little Arithmetic

PRINT is a very versatile and powerful BASIC instruction. Watch what happens when we run the following little program (first erase the previous program by entering NEW):

```
10 PRINT "3+16"  
20 PRINT 3+16  
30 END  
RUN  
3+16  
19  
OK
```

Line 10 does what you expected it to do: it just prints the string “3 + 16”. The surprise is what line 20 does: instead of printing what you typed, it actually adds 3 plus 16 and *prints the answer*. Evidently that happened because we left off the quotation marks that were previously used to identify a string. This example illustrates a general principle: if you follow PRINT with a simple arithmetic problem, the Model 100 will print out the *answer* rather than merely restate the problem.

Subtraction, multiplication, and division are just as easy as addition. The following example illustrates these four common arithmetic operations. Clear the current program with NEW and then enter the following program:

```
10 PRINT 2 + 3  
20 PRINT 2 - 3  
30 PRINT 2 * 3  
40 PRINT 2 / 3  
50 END
```

Here is the output of this program:

```
RUN  
5  
-1  
6  
,666666666666667 ← A fourteen-digit number  
OK
```

The symbols used in addition and subtraction are familiar ones. To indicate multiplication, use the asterisk (*), and to indicate division, use the slash (/). Table 3-1 summarizes the symbols used for the four common arithmetic operations.

There are several details to notice about the output of the above program. First, results that are positive numbers have one blank space in front of them; however, when a negative number is printed out, its negative sign (-) occupies the space that was left blank in the case of the positive numbers.

Printing Arithmetic Operations

PRINT can be used to obtain the results of arithmetic operations (addition, subtraction, multiplication, and division), as shown in the following example:

```
10 PRINT 5 - 2
RUN
 3
OK
```

These numbers are printed out with a space in front of the first digit reserved for the sign of the number. The sign is omitted but understood for positive numbers, but the negative sign is shown for negative numbers.

Another thing to notice about the output of this program is that while the result of dividing 2 by 3 is correct for most practical purposes, it isn't exact: an exact answer would show an *infinite* number of 6s! Because that is hardly practical, the people who wrote BASIC for the Model 100 decided to call it quits after fourteen digits. The fourteen-digit precision to which your Model 100 "defaults" is called *double precision*. (We'll discuss the precision of numbers later, but it's important to note here that BASIC for your Model

Operation	BASIC Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/

Table 3-1. Common arithmetic operations

100 calculates to an accuracy of fourteen digits — unless you instruct BASIC to do otherwise, using an instruction we'll show you later.)

The previous program printed out the results of numeric operations. What if you asked BASIC to print a simple number with no arithmetic operations performed on it? No problem: the output will look much like the printed output of the numeric operations in our original program. Try this example:

```
10 PRINT "4009"  
20 PRINT 4009  
30 END  
RUN  
4009  
 4009  
OK
```

Again, notice the difference between printing a number within a string and a number not within a string: the blank space reserved for the sign of a number is printed only if the number is *not* part of a string. Numbers not within a string, as well as numbers that result from arithmetic operations, are called *numeric expressions*.

You can also use PRINT to print numeric expressions in the Direct Mode. In this way you can make your not-so-inexpensive Model 100 emulate a \$20 calculator! That is a handy feature at times, and it does speak for the versatility of your computer.

Printing with Semicolons and Spaces

So far we have learned how to print either a single string or the result of a single calculation with each PRINT statement. However, a single PRINT statement can also be used to print a *series* of expressions, whether they are strings or numbers. Furthermore, you can control the format of the output somewhat by means of the following special characters: the semicolon, the space, and the comma.

The semicolon and the space (generated by pressing the space bar) do pretty much the same thing, so we will look at them together. The following illustrates the use of the semicolon and the space to control output:

```
10 PRINT "watch";"the";"spacing"  
20 PRINT "watch" "the" "spacing"  
30 END
```

```
RUN
watchthespacing
watchthespacing
OK
```

As this program shows, to print several strings one right after the other with one PRINT statement, simply separate the strings with a semicolon or a space. Note that the strings are really “stuck” together. In general, semicolons or spaces between strings cause their contents to be printed out in sequence without any spaces between them.

If you *do* want spaces, you have to put them into the strings yourself. For example, let’s change line 20 so that it returns something more readable by reentering line 20 to read:

```
20 PRINT "watch " "the " "spacing "
```

Note the space following each word and *within* the quotes. The output of our program now looks like this:

```
RUN
watchthespacing
watch the spacing
OK
```

So if you want your strings to be printed with spaces between them, you have to put them in yourself.

Printing a Sequence of Numbers

You probably have already guessed that we can also do with numbers what we’ve just done with strings. Try the following example. First, enter NEW, then enter the program as shown below, and finally enter RUN.

```
10 PRINT "1";"2";"-3";"-4";"5"
20 PRINT 1;2;-3;-4;5
30 END
RUN
12-3-45
1 2 -3 -4 5
OK
```

This example shows the difference between printing a sequence of strings and printing a sequence of numbers. Line 10 works pretty much the way we expected; it prints the contents of the strings without any spaces to give “12-3-45”. Line 20, however, has a surprise for us: it returns the proper sequence of numbers, but for some reason all those spaces have been added.

It turns out that there are two kinds of spaces involved. The first has to do with something you're already familiar with: BASIC always leaves room for the sign of the number, whether it is shown explicitly (for a negative number) or just implied (for a positive number). The second kind is a space following each number. As a result, each one-digit number takes up three spaces when printed. Figure 3-1 shows the output of line 20 with these spaces clearly marked by vertical lines. BASIC inserts this space following each number so that numbers that really are separate will not be run together.

Mixing Strings with Numbers

Strings and numeric expressions may be mixed in a single PRINT statement. In the following example, we'll use the PRINT statement in the Direct Mode for the sake of variety. Enter the BASIC command shown below (no line numbers!). Also shown is what BASIC returns immediately after **ENTER** has been pressed:

PRINT ",06 * 586.24 =" ; ,06 * 586.24	← What you enter
,06 * 586.24 = 35.1744	← What BASIC returns

First, the contents of the string are printed out, and immediately following the string comes the result of the multiplication. Nothing surprising really, but this example does show how you might calculate 6 percent interest on a \$586.24 purchase and do it using a rather nice format made possible by the PRINT statement.

Using PRINT with Semicolons or Spaces

A PRINT statement followed by a sequence of expressions separated either by a semicolon or a space prints out the sequence of expressions one right after another, without spaces. An expression may be a string or it may have a numeric value. Numeric values returned by PRINT always have spaces reserved for the sign of the number (implicit for positive numbers) and a space after the number in order to keep numeric output unambiguous.

Printing with Commas

Sometimes you may find it desirable to print strings or numbers in well-defined columns, as, for example, in producing a phone list or an inventory. You could do this by printing with semicolons and spaces inside strings, as discussed in the previous section, but this would be tedious, at best. You can directly and easily arrange your output into neat columns by using the PRINT statement and substituting a *comma* for the semicolon or the space.

To illustrate this use of the comma, let's enter the following new program (type NEW first) and run it:

```
10 PRINT "one","two"
20 PRINT "Phone number", "456-1308"
30 PRINT 1,-2
RUN
one           two
Phone number  456-1308
1             -2
OK
```

As you can see, the comma is responsible for arranging the output into *columns*. By counting spaces, you'll find that the second column begins with the fifteenth character in each line. Each column has a well-defined *left* edge; such columns are said to be *left-justified*. Notice that this left-justification works for the numerical output as well, provided that you remember there is always a space reserved for the sign of the number.

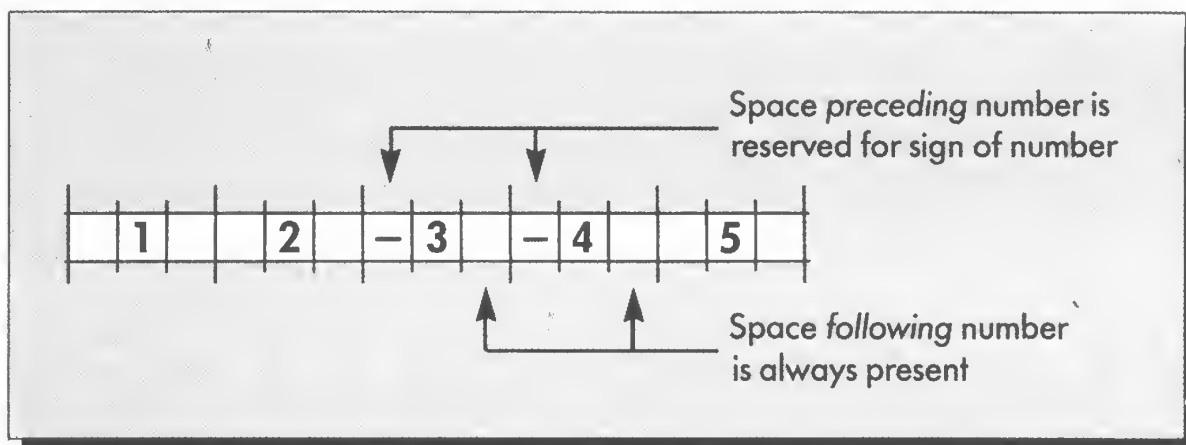
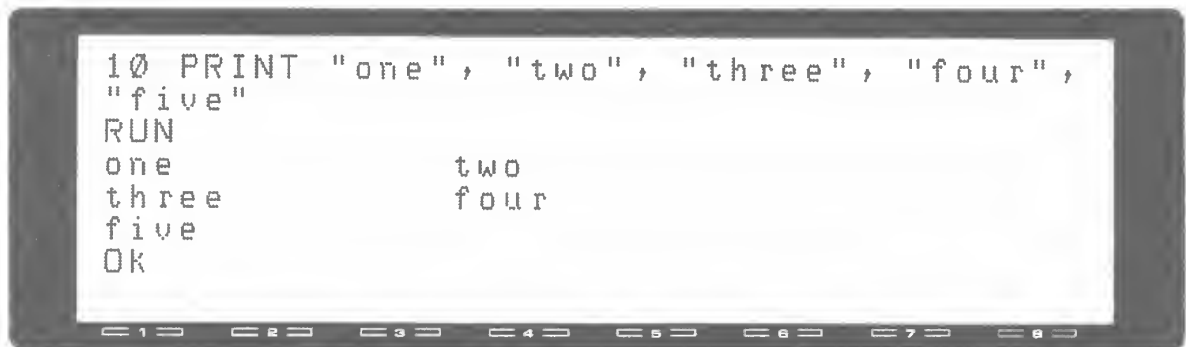


Figure 3-1. Output format of printed numbers using semicolons

Several questions come to mind at this point. First: "How many columns can we PRINT using commas?" To find the answer, enter NEW and try the following example:



```
10 PRINT "one", "two", "three", "four",  
"five"  
RUN  
one                two  
three             four  
five  
OK
```

The answer is clear: PRINTing with commas produces only *two* columns. If you list more than two items (a string or a number) to be printed, the third item is printed in the first column, the fourth item in the second column, and so forth.

The second question is: "What happens if a string or a number is too long to fit into either the first or second column?" The longest string that fits into the first column without causing the second string to be pushed out of the second column has thirteen characters. Consequently, there is always at least one blank space between the two columns to keep them from running together.

How many characters can you have in the second column? BASIC is more generous in this instance: you can have up to twenty-five characters in the second column, leaving only one space at the right edge of the screen.

Printing with Commas

A PRINT statement containing two expressions separated by commas prints the listed expressions in two left-justified columns. The first begins with the first character position on a line, and the second, with the fifteenth character position.

Printing with TAB — Controlling Your Columns

Using commas in your PRINT statement is certainly a convenient way to produce two neat columns of information. But this format may be too restrictive if you need more than two columns or a column spacing different from fourteen columns.

This is where the TAB instruction comes to the rescue. TAB in BASIC works very much like the tab key on your typewriter: it is a convenient, quick way to write out something at a specified character position. The required format of the TAB statement is TAB(*n*), where “*n*” stands for the number of spaces (from 0 to 255) before the following string or number is printed. To get a sense of how TAB works, enter and run the following program:

```
10 PRINT "01234567890123456789012345"
20 PRINT "1" TAB(10) "2" TAB(20) "3"
30 PRINT "Groucho" TAB(10) "Harpo" TAB(20) "Zeppo"
30 END
RUN
01234567890123456789012345
1          2          3
Groucho    Harpo      Zeppo
OK
```

Line 10 is included for the sole purpose of supplying a “ruler” to help us “measure” character positions. (This ruler is only a convenience; it is not required to use TAB.) The first character position on this ruler is labeled “0”, and we count upward from there. (We could just as well begin with “1” rather than “0”, but the ruler presented here has the virtue of being consistent with character positions we’ll use with the PRINT @ statement in the next section.) Because we need to indicate each character position with a single digit, we dropped the first digit of all character positions larger than 9; thus, position 10 is indicated by a “0”, 11 by a “1”, and so forth.

As you can see from the output of our program, the number inside the parentheses following the word *TAB* determines the character position where the string (or number) listed immediately after TAB is PRINTed out. The absence of the TAB function between PRINT and “1” in line 20 is equivalent to TAB(0). TAB(10) in line 20 causes the string “2” to be PRINTed in the character position 10; similarly, TAB(20) causes “3” to be printed in character position 20. Although the number within the parentheses after TAB can be as large as 255, the right-most position on the first line that we can PRINT on is character position 39 (which would be position 40 if we’d begun our ruler with 1 instead of 0).

Note that we used TAB within a PRINT statement. We cannot use TAB by itself; we must use it with PRINT or with the related statement LPRINT, which we'll discuss later. TAB's utility is obvious: it is a simple way to print information anywhere on a line; in particular, TAB makes it easy to define the left edge of columns.

Printing with TAB

The BASIC statement

```
20 PRINT TAB(5) "Space"
```

means “print the string ‘Space’ so that the first character of the string (‘S’) appears at the fifth character position of your screen”. The first character position is taken to 0, with 39 being the last on one line.

The PRINT @ Statement — Better Control over Your Screen

So far you have learned to use PRINT to print whatever you want anywhere on the screen. Additionally, you can get to any horizontal position by using the TAB function, and you can get to any vertical position by using blank PRINT statements (PRINT without anything following it). But there is one thing you can't do yet: namely, move back up the screen and print at a location above where you've printed before. Also, to get from the middle of the screen to some location near the bottom, you would have to use a lot of blank PRINT statements.

The PRINT @ statement is designed to solve these problems. It's similar to the PRINT statement but has the added capability of letting you print something anywhere on your screen with a single statement.

To use the PRINT @ statement, we need a way to specify the position that a character (a letter, for example) can occupy. On your Model 100, the location of a given character space is determined by a single number that we'll call the *character coordinate*. The character coordinate is similar to the number on your house or apartment — it tells you exactly where you can locate something. Figure 3-2 shows the character coordinates of selected character positions on your screen.

As you can see, the first row starts with character coordinate 0 and ends with character coordinate 39; similarly, the second row begins with 40 and ends with 79. It's easy to see how these numbers are obtained: simply count characters line by line as if you were reading a page full of text, remembering, however, that our count begins with 0. Because your screen has a width of forty characters, the beginning of each row is specified by multiples of forty.

How do you use these character coordinates to determine where you want to PRINT something? Try the following new program:

```
10 CLS
20 PRINT @137, "center"
OK
```

Here is the output as it appears on the screen:



The number after the @ sign in line 20 of the program is the character coordinate we've been talking about. The effect of line 20 is to print the

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60											70									79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100											110									119
120	121	122	123	124	125	126	127	128	129	130										140											150								159	
160	161	162	163	164	165	166	167	168	169	170										180											190							199		
200	201	202	203	204	205	206	207	208	209	210										220											230							239		
240	241	242	243	244	245	246	247	248	249	250										260											270							279		
280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	

Figure 3-2. Character coordinates on your Model 100 screen

word *center* in such a way that the first letter of the string (*c*) appears at character coordinate 137. The following diagram identifies the various ingredients in our PRINT @ statement:

```
20 PRINT @137, "center"
```

String (or number) to be printed

Character coordinate that specifies where the first character of the string (or number) is to be printed

How did we arrive at the rather peculiar number 137 for our character coordinate? We wanted to place the word *center* near the physical center of the screen, which is located between rows 4 and 5. We chose the fourth row, beginning with coordinate 120; so we figured that the middle of the word *center* had to be at coordinate 120 plus 20 — half the screen width — or 140. Because the word *center* has six letters, we subtracted half that amount (3) from the coordinate of the screen's center (140) to get the result of 137.

Our next example further explores this important and useful variation of the PRINT statement. The following program places an "x" at the corners of your Model 100 screen:

```
10 CLS
20 PRINT @0, "x"
30 PRINT @39, "x"
40 PRINT @280, "x"; ← Semicolon to suppress carriage return
50 PRINT @318, "x"; ← Semicolon to suppress carriage return
60 PRINT @120, ""
OK
```

Before we analyze the details of this program (which contains some surprises!), let's look at the output shown below:



Several aspects of this program are not immediately obvious, beginning with the semicolons at the end of lines 40 and 50. Semicolons at the *end* of PRINT @ statements suppress unwanted carriage returns. If we omit the semicolon from, say, line 40, the carriage return that is automatically executed at the end of a PRINT statement will cause the screen to scroll upward one line; the result is that the “x”s we so carefully placed at the upper corners will be scrolled into oblivion.

A second feature of this program that may be puzzling is that the last “x”, which was to be placed in the lower right corner of the screen by line 50, is one space shy of the corner! When we tried to place the “x” *exactly* in the corner by using the character coordinate 319 (rather than 318, the coordinate used in the program as listed above), an “x” appeared briefly at the expected position but, alas, was quickly scrolled up to the next line. In this instance, we cannot suppress the carriage return with a semicolon. The most likely explanation for this apparent anomaly is related to the fact that this corner is the very last character coordinate on the screen.

The last feature of our program that needs clarification is the placement of the string “” at coordinate 120 by line 60. The purpose of this line is to move the BASIC prompt and the cursor away from the lower boundary of the screen. If we omit this statement, the BASIC prompt and the cursor, which usually appear right after the last PRINTed line, will cause the screen to scroll upward two lines, and our “x”s will again be moved out of the corners. Here’s how line 60 works. First, notice the absence of anything between the two quotation marks in the string “”. This string is called a *null string*, that is, a string with a “null” or zero content. When line 60 is executed, nothing is printed by the PRINT statement. However, the BASIC prompt and the cursor following it behave as if a normal string had been printed; that is, the cursor moves to the line *below* the line on which the null string was PRINTed. Since we printed the null string on line 4 (character coordinate 120), the BASIC prompt appears on line 5.

The PRINT @ statement is important and extremely useful whenever you need to place strings or numbers at specific locations on your screen. Later in this book, we’ll have many occasions to use it in formatting output for serious applications or in creating patterns of special characters for game-related programs.

The PRINT @ Statement

We can use the PRINT @ statement to PRINT a string or number anywhere on the screen by specifying the character coordinate as illustrated in the following example:

```
40 PRINT @137, "piccolo"
```

This statement prints the word *piccolo* at the screen coordinate 137, located near the center of the screen. (See Figure 3-2 for a listing of screen coordinates.)

Summary

In this chapter we have explored the PRINT statement, and we have seen how TAB can be used within a PRINT statement to determine the horizontal position at which a string or number is to be printed. In addition, we've shown how you can print something *anywhere* on your Model 100 screen by using a single PRINT @ statement.

As a way of reviewing this chapter, we suggest that you reread the boxed summaries. Then test your newly acquired skills by writing the programs suggested below in "Exercises". These programs involve most of the BASIC statements covered in this chapter.

If it seems that printing and doing arithmetic with a computer is little improvement over the typewriter and calculator, keep in mind that this chapter mainly concerned *output*. We haven't really touched on what computers do best — namely, doing manipulations or calculations over and over again — the kind of work that humans find extremely tedious. So hang on, you have much to look forward to.

Exercises

1. Write a program that clears the screen, prints out the centered title "MY SHOPPING LIST", then lists three shopping items with prices, and finally prints out the total cost.
2. Write a program using the PRINT @ statement that places a small cross near the center of your screen. Construct the cross out of asterisks.

Solutions

1. The following is our solution; yours may differ in some of the details.

```
10 CLS
20 PRINT TAB(12) "MY SHOPPING LIST"
30 PRINT
40 PRINT "ITEM: cat food,          $5.86"
50 PRINT "ITEM: can of sardines,    $1.58"
60 PRINT "ITEM: Model 100 batteries,$5.98"
70 PRINT "TOTAL = $"; 5.86+1.58+5.98
```

When you RUN this program, the following appears on your screen:



2. The following program prints a cross of asterisks near the center of the screen:

```
10 CLS
20 PRINT @139, "****"
30 PRINT @100, "*"
40 PRINT @180, "*"
50 END
```

The output looks like this:



Using this model, you can easily produce any kind of pattern on your screen. We'll use this technique in more sophisticated applications later on.

4

Housekeeping Chores

Concepts

- Special function keys
- Programming programmable function keys
- Editing with EDIT
- Comments within a program
- Using a printer

Instructions

- The special function keys **F1** through **F8**
- The command keys **PRINT** and **PAUSE/BREAK**
- BASIC instructions **CONT**, **FILES**, variations of **RUN**, **LIST**, **REM**, **LCOPY**, **LLIST**, and **LPRINT**

The Model 100 computer has many features that simplify and facilitate writing BASIC programs, including a set of programmable function keys, which simplify the entry of frequently used commands; variations of the **LIST** and **RUN** commands, which make it easier to write and to correct (or *debug*) programs; the **REM** statement, which enables us to write explanatory comments into our programs about how the program works; an editor, called **EDIT**, which facilitates making changes in programs; and, finally, a collection of instructions that allows us to use a printer to list and produce “hard-copy” program output.

In this chapter we discuss these features and their functions. You’ll find this chapter an important aid to programming BASIC on your Model 100.

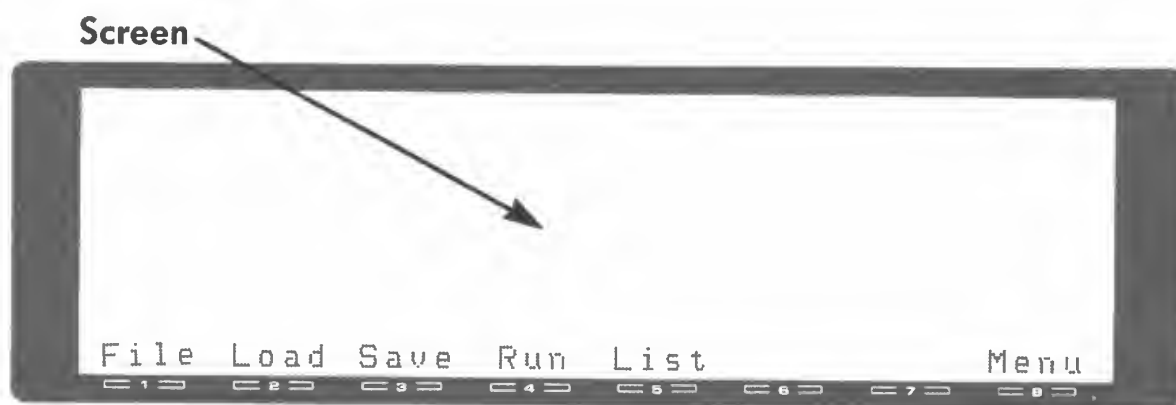
Special Keys — Making the Programmer’s Life Easier

Let’s start by exploring the special keys located above the typewriter part of your keyboard. (Figure 4-1 shows their location and how they’re labeled.)

These keys fall into three categories: *programmable function keys*, *command keys*, and *cursor control keys*. We'll introduce these keys and their functions in the following sections, although a thorough explanation of certain keys will have to wait until later, when we have acquired the necessary background.

Programmable Function Keys

The first set of special keys comprises the eight programmable function keys. These are the left-most eight keys, labeled (F1) through (F8). To find out what they do, turn on your Model 100 and get into the BASIC Command Mode by selecting BASIC from the menu. Then press the (LABEL) key and note what appears on the bottom line of your screen:



The word that appears on the screen above each function key defines the purpose of the key. Of the words *File*, *Load*, *Save*, *Run*, *List* and *Menu*, you are already familiar with *List*, *Run* and *Menu*. The purpose of these function keys is to make it easier and faster for you to enter frequently used commands. Pressing a programmable function key takes the place of entering the corresponding BASIC command and pressing (ENTER). For example, instead of entering LIST by typing "LIST" and then pressing (ENTER), you can simply press the key (F5) — try it. (Unless you've entered NEW after the last chapter, "MY SHOPPING LIST" is your present program.) Also, instead of entering RUN by typing "RUN" and pressing (ENTER), you can simply press the key (F4).

The remaining words associated with the function keys are new to us. Here we'll briefly identify what they mean and what the related function keys do. "File" identifies the key (F1). When you press (F1) your Model 100 will list the names of all its stored files — BASIC programs and text that has previously been stored in the computer's memory by a method we'll describe in the next chapter. The next two keys, (F2) and (F3), also have to do with files. The key (F2), labeled "Load", helps us recall a BASIC pro-

gram previously stored in the computer's memory, and the key (F3), labeled "Save", enables us to store a BASIC program in the computer's memory as a file. The key (F8), labeled "Menu", causes your Model 100 to return from the BASIC mode to the main menu.

Notice that keys (F6) and (F7) aren't labeled. That tells us that these keys are not presently active; when you press them, nothing will happen. We said not *presently* active because we can *program* these keys to perform various functions — hence the name *programmable* function keys. All the keys except (F6) and (F7) have already been programmed for you.

Programming Your Programmable Function Keys

Any of the programmable function keys can be programmed to execute a BASIC command of your choice, even the keys that have already been programmed. To illustrate, let's program the (F6) key to display the date. Many aspects of what we'll show you won't really make sense until later, when we talk about the ASCII code and string functions; so don't worry if you don't understand everything at this point. Perhaps you can use this example as a model for programming these keys to suit your own needs; if not, feel free to skip this section — it's not essential. You can always come back to it when you have a bit more BASIC under your belt.

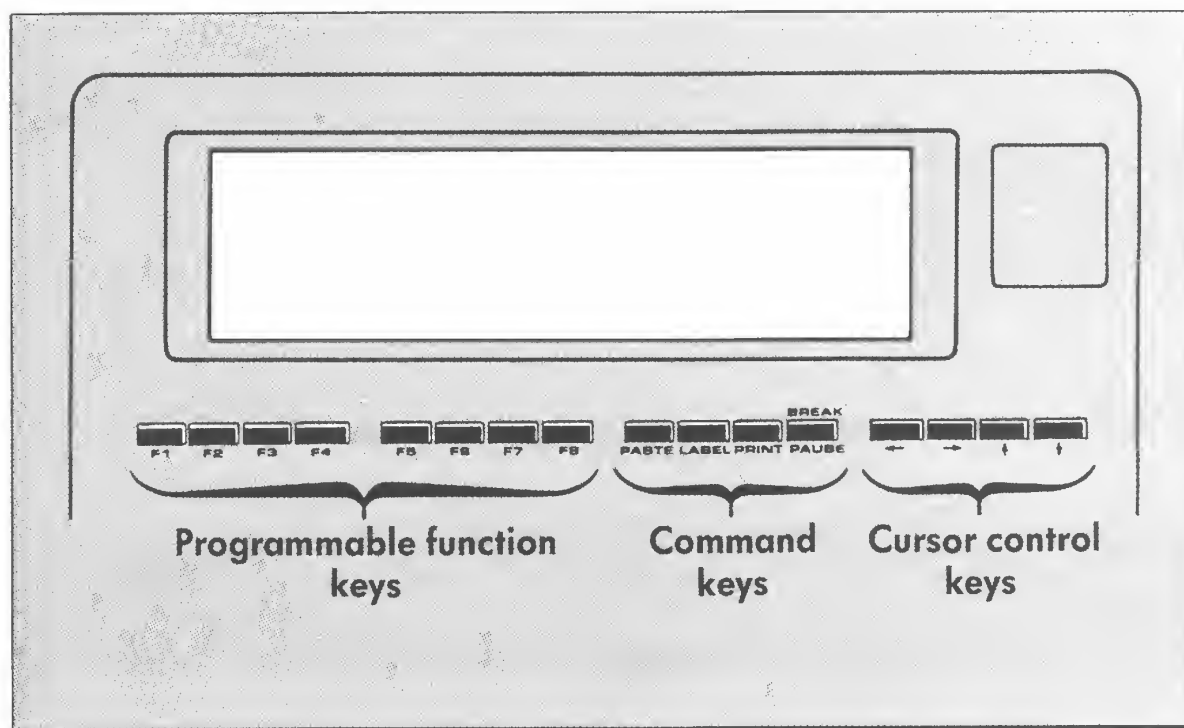


Figure 4-1. Special keys of the Model 100

To program the key **F6** to give you the date, enter the following line (we assume you're still in the BASIC Command Mode, that is, you've got the Ok sign and the blinking cursor on your screen):

```
KEY 6, "?DATE$" + CHR$(13)
Ok
```

The characters "?dat" immediately appear on your screen right above the **F6** key. Because there is room for only four characters, the letter *e* is left off the word *date*. What happens when you now press the key **F6**? Try it. The instant you press **F6**, the date will appear on your screen:

```
Ok
?date$
10/24/84
Ok
```

Great! Now, every time you press **F6**, the date appears on your screen. Though we won't now explain all the details of the instruction that programmed the **F6** key, we will summarize what each part of the command does. The first part of the entered command (KEY 6) tells your Model 100 that you're about to give the key **F6** a new definition or function. The next part, after the comma ("?DATE\$"), specifies what this function is. The first character within the quotation marks is a question mark, which is an abbreviation for PRINT. You can use a question mark in place of PRINT in *any* of your BASIC programs. The whole phrase within the quotation marks means "print the date". The last part of the reprogramming instruction for key **F6** (+ CHR\$(13)) has the same effect as pressing the **ENTER** key. One important general rule is that the instruction after the comma (right after KEY 6 in our example) must be equal to or less than fifteen characters long, not counting spaces or quotation marks.

To summarize, the command KEY 6, "?DATE\$" + CHR\$(13) tells your Model 100 to print the current date whenever the key **F6** is pressed. You can program and reprogram any of the programmable function keys in a similar manner. You may wish to program the **F7** key to give you the *time* by substituting the BASIC word TIME\$ for DATE\$ and **F7** for **F6** in our above example; the complete command is shown below:

```
KEY 7, "?TIME$" + CHR$(13)
```

Pressing the key **F7** will now give the time on your screen.

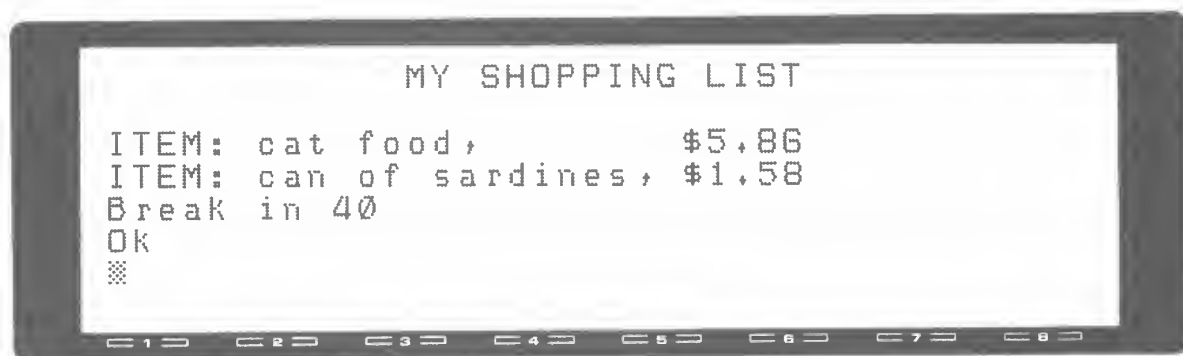
The Command Keys

The four keys to the right of the eight programmable function keys are called *command keys*. These are not programmable but are designed to perform frequently needed functions. The first two are **PASTE** and **LABEL**, which are used to move blocks of text when you are using the built-in program TEXT. Though they also have some limited use when editing BASIC programs (in the BASIC editor called EDIT), they are of little practical importance in writing BASIC programs, so we won't explain how to use them in this book. They are explained thoroughly in the book in this series called *Introducing the TRS-80® Model 100* by Diane Burns and S. Venit (New York: Plume/Waite, New American Library, 1984).

The third command key is labeled **PRINT** and is functional only if you have a printer attached to your Model 100. If you do, you'll find this key to be extremely useful. Its purpose is to cause your printer to print out whatever appears on your Model 100 screen. We'll further discuss this key later in this chapter, along with several other printer-related instructions.

The fourth command key, **PAUSE/BREAK**, has two labels: PAUSE below the key and BREAK above it. Pressing this key alone causes any program that is presently *running* to *pause*, that is, to be temporarily interrupted. It is sometimes useful to interrupt a program in this manner in order to view part of a long program output before it scrolls out of sight. The programs we've written so far have been fairly short, so you'll have to be quick to press **PAUSE** *after* entering RUN — but *before* the program is finished running! Pressing **PAUSE** a *second* time causes your Model 100 to continue executing your current program. **PAUSE** also temporarily interrupts LIST, enabling you to “freeze” and view parts of long programs. **PAUSE** is a *toggle* switch: when pressed once, it is on — causing a pause — and when pressed again, it is off — resuming execution.

Pressing the **PAUSE/BREAK** key *while holding down the* **SHIFT** key (on the typewriter part of your keyboard) activates the uppercase, or BREAK, function of this key. From now on we'll identify this combination of keys as **SHIFT BREAK**. Pressing these two keys will cause a program that is currently being executed by your Model 100 to be interrupted and your BASIC prompt to reappear. For example, if you run the program “MY SHOPPING LIST” (from the previous chapter) and you press **SHIFT BREAK** while the program is still being executed, something like the following will appear on the screen:



The message “Break in 40” tells you that program execution was interrupted at line 40. If you’re very quick, you can interrupt it at line 5. If you miss the boat altogether — the program finishes before you press **BREAK** — the symbol “^ C” appears on your screen, which simply indicates that the **SHIFT** **BREAK** keys have been pressed.

Although **SHIFT** **BREAK** returns your Model 100 to the BASIC Command Mode, program execution is not irrevocably terminated. If, after pressing **BREAK**, you enter the BASIC command **CONT** (for “Continue”) before you enter any others, your program will continue executing from the place where it was interrupted by **SHIFT** **BREAK**.

Because the programs we’ve written so far are fairly short, the utility of the **PAUSE/BREAK** key is not yet obvious. However, as we write longer and more sophisticated programs, you’ll find the **PAUSE** function invaluable when you want to look at program output or at particular portions of a program listing. You’ll find the **BREAK** function useful in breaking or terminating program execution when you don’t like what’s going on!

The last four keys, labeled only with arrows underneath them, are *cursor control keys*. These function only in the **TEXT** program or in the BASIC editor **EDIT**, which we’ll discuss now.

Editing in the EDIT Mode

In Chapter 2 we learned how to use the **DEL/BKSP** key to make corrections in a program line before **ENTER** has been pressed. We also saw how to change existing program lines (reenter the whole line) and how to add and delete lines (enter the desired line prefixed with the appropriate line number). With these techniques you can make any needed changes in your BASIC programs.

However, there is another way to correct program lines that is often more efficient and convenient. This is a special program incorporated into BASIC called **EDIT**. (If you are familiar with **TEXT**, you’ll have a headstart: **EDIT** is an editor in BASIC that provides the same editing tools as those

provided by TEXT, an editor for text files.) By using EDIT you can correct an existing program line (a line that you've entered) without retyping the whole line — something you cannot do in the BASIC Command Mode. Hence EDIT is particularly useful in editing long program lines.

For example, suppose we've entered the following BASIC lines:

```
10 PRINT "Happy, Silly, Grumpy,"
20 PRINT "Doc, Dopey, Sleepy"
OK
```

You may now realize that the name Bashful is missing and decide to include it between Doc and Dopey in line 20. You *could* retype the whole line and press **ENTER**, but that is a bit tedious and provides an opportunity to make more mistakes. It is easier to use EDIT. In the following sections we describe the use of EDIT to make this and similar corrections.

Getting into the EDIT Mode

First we need to call up the editor EDIT. From BASIC, enter the word *edit* — type the word edit in lowercase or uppercase and press **ENTER** — and the following will appear on your screen:



You are now in the EDIT Mode. Notice that the BASIC prompt Ok is missing; instead, there are two new symbols. One is a triangle that looks like this: ◀. It marks the end of your BASIC lines (the end of the logical line) and represents **ENTER**. In the BASIC Command Mode, you can't see where you've pressed **ENTER**, but in the EDIT Mode, **ENTER** is made visible by this triangle. The other new symbol is an arrow that points to the left (←). This arrow marks the end of the text. We'll see how it moves around later, as we enter new text.

Although the BASIC prompt is absent, the blinking cursor is still with us in the upper left corner, superimposed on the number 1. As in the BASIC Command Mode, the cursor tells us where a character will appear on the screen if we press a character key; so if we pressed the “m” key, the letter *m* would appear in the upper left corner.

Moving Your Cursor

Well, that’s *not* the place where we want to make our correction. We want to insert the name Bashful on the *second* line, right after the name Doc. To make a correction (an insertion in this case), we must first move the cursor to the correct position on the screen. By now you’ve probably guessed that the four keys with the arrows underneath them at the upper right on your Model 100 have the function of moving the cursor. Go ahead and use them to move the cursor all over the screen; the text itself will not be affected. Notice that you can’t move the cursor below the third line containing the arrow that marks the end of the text. The cursor is confined to the existing text and the space on the same line as the arrow. Figure 4-2 shows the cursor control keys and their functions.

Although these four cursor control keys allow us to move the cursor anywhere within a program, various other methods of moving the cursor may at times be more convenient. One method is to use the cursor control keys while holding down either the **SHIFT** key or the **CTRL** key (for “ConTRoL”) on your keyboard. These combinations are useful in moving the cursor rapidly throughout large programs. Table 4-1 summarizes the effects of these particular key combinations.

Inserting Characters

To insert “Bashful”, place the cursor at the position where you want to place the first character — that’s the *D* of Dopey. Now type in “Bashful”. As you type the letters *B*, *a* and so forth, the cursor moves to the right, taking with it the letter superimposed on it and the text to the right of it. In other

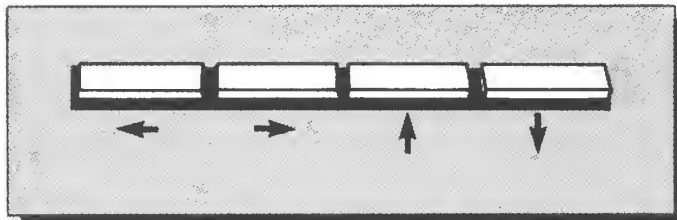


Figure 4-2. Cursor control keys; arrows indicate direction of cursor movement

words, insertion takes place automatically: you can't lose any text by trying to type right over something because the old text moves over to make room for the new text. Figure 4-3 summarizes how this insertion works.

Replacing and Deleting Characters

If you're familiar with *Snow White and the Seven Dwarfs*, you'll realize that there is still something wrong with our collection of names: the second name in line 10 should be Sneezy, not Silly. How do we replace Silly with Sneezy? Replacing characters is a two-step process: first you delete the unwanted characters, and then you insert the new characters. The order doesn't matter; you can do it the other way around just as well. Since we already know how to insert characters, we'll now describe how to make deletions.

One way to delete a character is with the **DEL/BKSP** key. To use this key to erase the name Silly, place the cursor to the *right* of the word *Silly*, right on top of the comma; then press the **DEL/BKSP** key as many times as you need in order to erase the whole word (five times to erase Silly). Notice that each time you press **DEL/BKSP**, the cursor, the character under it, and everything to the right of it move to the left one space. As you erase a character, the text to the right of it moves in to replace the erased character. Figure 4-4 summarizes the effect of the **DEL/BKSP** key.

The second method of erasing something in EDIT is to use the *uppercase* function of the **DEL/BKSP** key. As you know, when you press **DEL/BKSP** alone, the character to the left of the cursor is deleted. However, if you hold down the **SHIFT** key and *then* press the **DEL/BKSP** key, the character *under*

Key Combination	Effect
SHIFT + →	Moves cursor to beginning of word to immediate right
SHIFT + ←	Moves cursor to beginning of word to immediate left
SHIFT + ↑	Moves cursor to the top of the screen
SHIFT + ↓	Moves cursor to the bottom of the screen
CRTL + →	Moves cursor to right end of line
CRTL + ←	Moves cursor to left end of line
CRTL + ↑	Moves cursor to beginning of file
CRTL + ↓	Moves cursor to end of file

Table 4-1. Moving the cursor with the help of the **SHIFT** and **CRTL** keys

the cursor will be erased. This is a convenient way to delete a character in the middle of a line. To try it out, insert the name Silly back in its original position so that you have something to erase (you get a chance to practice inserting!). Now place the cursor right on top of the first letter of the word you want to erase (in our example, the letter S). Then press the **SHIFT** **DEL/BKSP** key (press the **DEL/BKSP** key while holding down the **SHIFT**

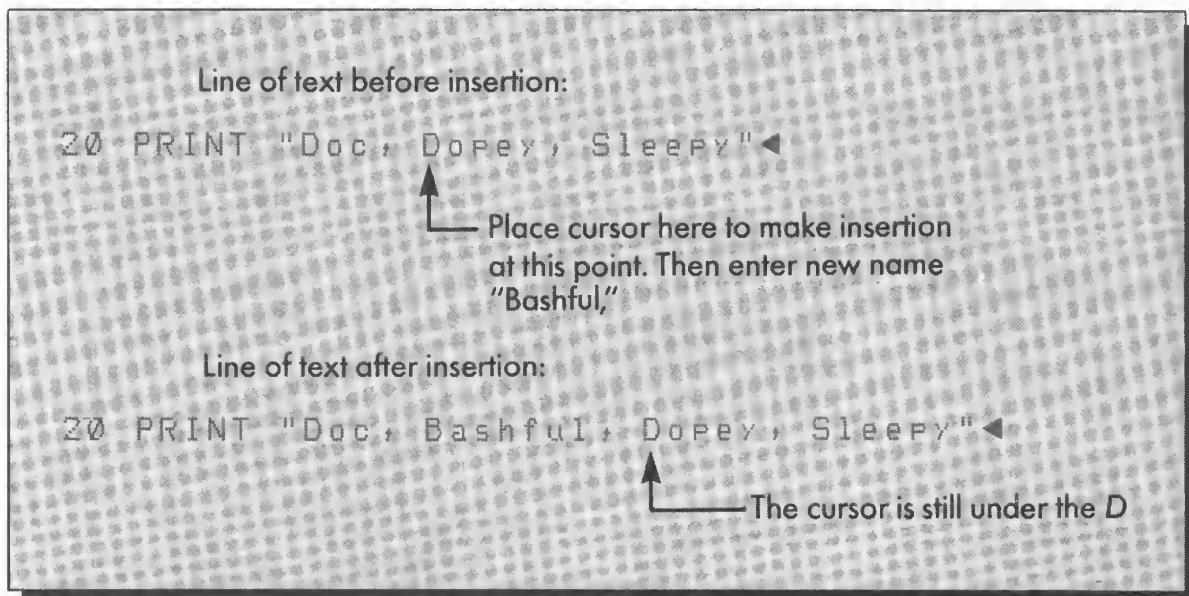


Figure 4-3. Inserting a word in EDIT

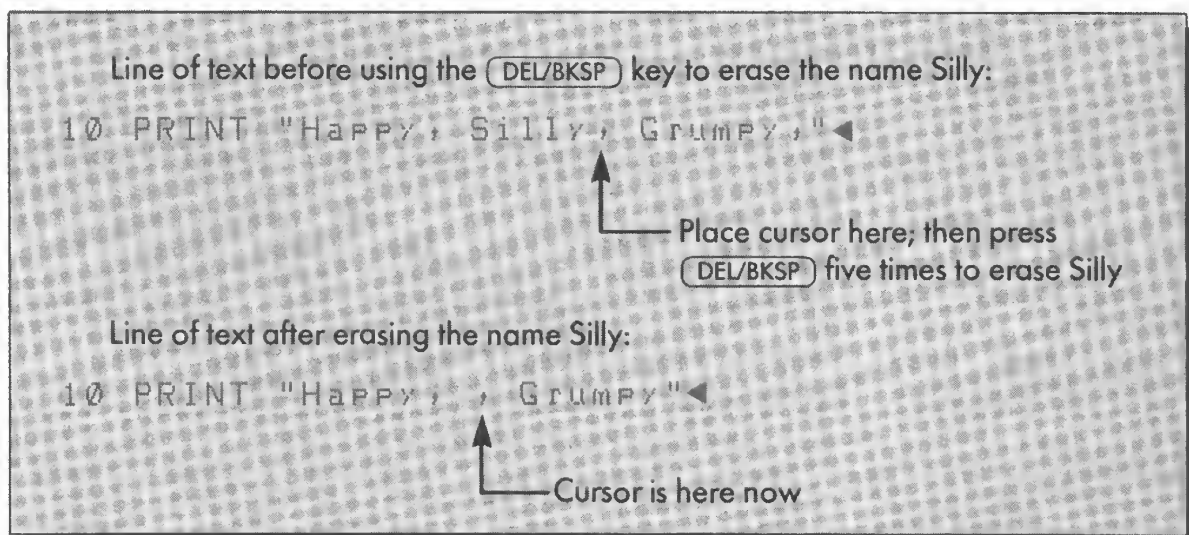


Figure 4-4. Use of the **DEL/BKSP** key in EDIT

key) five times. Each time you press **(SHIFT) (DEL/BKSP)**, the cursor remains fixed, but a character moves in from the right to replace the character originally superimposed on the cursor. The end result is the same as you obtained by using the lower case function of the **(DEL/BKSP)** key. Figure 4-5 summarizes the effect of using **(SHIFT) (DEL/BKSP)** to erase the name Silly.

Our original intention was to replace the name Silly with the correct name, Sneezzy, so we still need to insert the new name. That's easy: we need only type the name Sneezzy and we're done! Remember, typing something with the cursor in the middle of the text *inserts* rather than types over the original text.

Dividing, Joining, and Inserting Lines

Now that we've covered the functions most useful in editing BASIC programs, we'd like to mention a few other procedures you might occasionally find useful.

Suppose you wish to break a long line into two lines. That's easy. Just place the cursor at the position where you want to divide the lines and press **(ENTER)**. The character originally superimposed on the cursor and everything to the right of the cursor will appear on the *next* line. Chances are, though, that you'll need to make corrections and additions to *both* lines to make them into legitimate BASIC statements.

To *join* two lines, place the cursor on the **(ENTER)** symbol (**◀**) of the first line and press **(SHIFT) (DEL/BKSP)** (hold down **(SHIFT)** and press **(DEL/BKSP)** in order to erase **(ENTER)**). The second line will automatically append itself

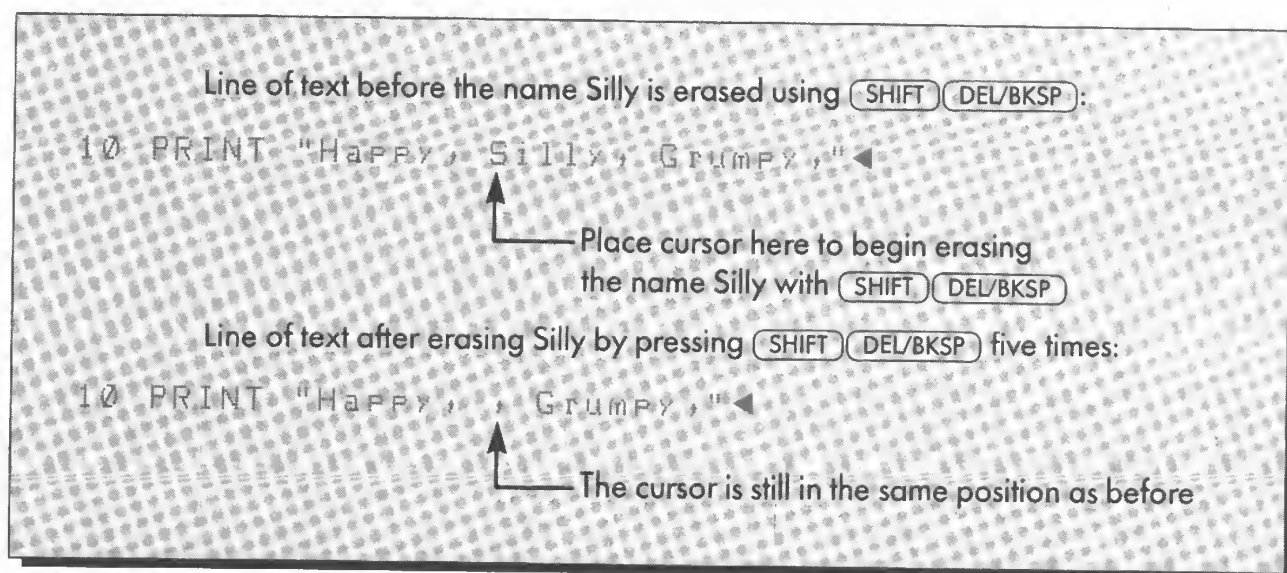


Figure 4-5. Using **(SHIFT) (DEL/BKSP)** in EDIT to erase a word

to the first. Notice that you can join two lines to undo what you've done — perhaps by mistake — with the division procedure discussed above.

You can also add a new BASIC line to your program within EDIT. Suppose you want to add the new line

```
5 PRINT "The seven dwarfs are:"
```

You can do it as you would in BASIC Command Mode: simply enter the new line at the end of your program. Place the cursor so that it is superimposed on the arrow marking the end of the text; then start typing and press **ENTER** when the program line is finished. Line 5 will appear at the *end* of your program just as it would if you added the line in BASIC Command Mode. When we get back into BASIC Command Mode, BASIC will, as always, rearrange the program lines according to line numbers.

Leaving the EDIT Mode

How do we get out of EDIT Mode and back to the BASIC Command Mode? Press the programmable function key **F8**, and after a brief wait you'll see the familiar BASIC prompt, **OK**. Now list your program to see that everything is as it should be. You get the following result:

```
LIST
5 PRINT "The seven dwarfs are:"
10 PRINT "Happy, Sneezy, Grumpy,"
20 PRINT "Doc, Bashful, Dopey, Sleepy"
OK
```

Yes, it looks like they're right this time!

Editing Long Programs

The program you edited with EDIT is a very short one that fits easily onto your screen. But how do you edit programs longer than one screen length? Several techniques can be used.

One method is to invoke EDIT as you did before by entering EDIT (assuming you were originally in the BASIC Command Mode). The screen limits the display to only the first eight lines of your program. What if you want to make changes in lines occurring after these first eight lines? Here's what you do: first, move the cursor down to the last line on your screen, and then press the downward cursor control key again. The cursor does not disappear below the lower edge of screen, as you might have expected; instead, the whole display on your screen scrolls upward! Every time you press the downward cursor control key, a new line appears at the bottom of

your screen. In this manner, all parts of your program are accessible for modification.

Another way to modify long programs is to transfer into EDIT only that part of the program you intend to edit. In the beginning of this section, you typed "edit" (in the BASIC Command Mode) in order to get into EDIT; and the *whole* program appeared on the screen. If, however, you now type

```
edit 10
```

the only line that appears on your screen in the EDIT Mode is line 10. That's also the only line you can edit. The command EDIT in the BASIC Command Mode followed by a line number displays only the line having that line number.

Other variations of the EDIT command allow you to call into the EDIT Mode different parts of a program. These are summarized below.

Using EDIT

1. To enter the EDIT Mode, use one of the following commands:

EDIT	Allows you to edit <i>whole</i> program
EDIT 50	Edit only line 50
EDIT 50 – 80	Edit lines 50 through 80
EDIT 50 –	Edit line 50 to end of program
EDIT – 230	Edit from beginning of program to line 230

2. To *insert* a character, place cursor to the *right* of the location where the character is to be inserted and simply type the character.

3. To *delete* a character, either

- a. place cursor over the character to be deleted and press **SHIFT** **DEL/BKSP**, or

- b. place cursor to the immediate *right* of the character to be deleted and press **DEL/BKSP**.

4. To exit the EDIT Mode to return to the BASIC Command Mode, press function key **F8**.

LIST and RUN Revisited — Getting More Particular

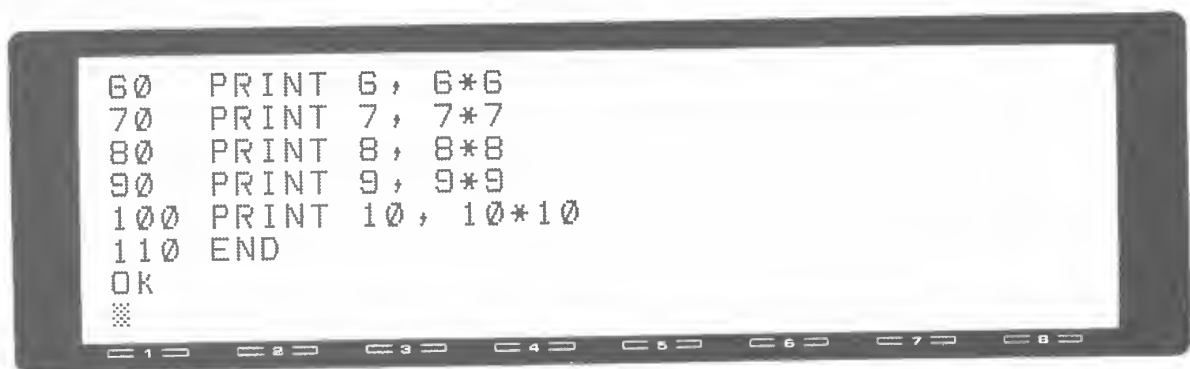
You already know how to use the LIST and RUN commands, but there are some variations of these commands we haven't yet introduced. These are not absolutely essential to programming in BASIC, but they can help save time in writing and debugging, especially on longer programs.

Extending LIST

To illustrate the problem of working with longer programs, consider the following program (if you already know some BASIC, you'll recognize that there is a better way to write this program; but all we need for this example is a program longer than six lines):

```
10 PRINT 1, 1*1
20 PRINT 2, 2*2
30 PRINT 3, 3*3
40 PRINT 4, 4*4
50 PRINT 5, 5*5
60 PRINT 6, 6*6
70 PRINT 7, 7*7
80 PRINT 8, 8*8
90 PRINT 9, 9*9
100 PRINT 10, 10*10
110 END
```

To look at what you've done, enter LIST. The beginning of the program appears first and then scrolls upward leaving the last six lines of your program:



You've got the last part of the program on the screen, but the beginning of the program has scrolled out of sight! What if you really wanted to look at line 50, for example? You *could* list the program and press the **PAUSE** key when line 50 appears on the screen, but that technique requires that you

watch the screen very carefully while your program scrolls by. Furthermore, you must *wait* until line 50 comes into view before you press **PAUSE**. Neither of these difficulties is very serious in the above example; however, if you want to look at line 150 in a program that ends at line 330, you'd probably be interested in another way to inspect a particular program line. The solution is to use the LIST command with some additional instructions (sometimes called "modifiers"). These instructions specify the line number or numbers to be listed; for example, the direct command LIST 50 causes only line 50 to be listed:

```
LIST 50
50 PRINT 5, 5*5
OK
```

No more waiting for the first four lines to scroll by, and no need to press the **PAUSE** key!

There are other ways to specify lines to be listed; for example, you can indicate a *range* of line numbers after LIST:

```
LIST 50-80
50 PRINT 5, 5*5
60 PRINT 6, 6*6
70 PRINT 7, 7*7
80 PRINT 8, 8*8
OK
```

As you can see, the modifier 50-80 causes LIST to list lines 50 – 80. Other variations of the modifier to LIST allow you to list program lines from the beginning of a program to a specified line number or from a specified line number to the end of the program. Table 4-2 summarizes all the variations of LIST.

Command	Function
LIST	Lists the entire program
LIST 50	Lists only line 50
LIST 50 – 80	Lists lines 50 through 80
LIST – 50	Lists from beginning of program to line 50
LIST 50 –	Lists from line 50 to end of the program
LIST .	Lists the line that was last listed, edited, or entered

Table 4-2. Variations of the LIST command

Extending RUN

Another useful aid in working on long programs is a variation of the RUN command. As you know, entering RUN causes your Model 100 to execute your present program. Execution starts at the first program line and proceeds to the last line. However, BASIC allows you to append the word *RUN* with a number that specifies the first line number to be executed. For example, if you enter RUN 80, the output of your previous program will appear as follows:

```
RUN 80
 8      64
 9      81
10     100
OK
```

The first line of the output is the number 8 and its square, 64, which is the output of line 80. BASIC then continues program execution in the usual manner until it reaches the END statement or runs out of lines to execute.

This variation of the RUN statement is useful if you're checking the last part of a long program. For example, if you corrected a mistake in line 80 in your program by reentering the line or using EDIT, you'd want to see if it now "works". Instead of running the whole program, you could run only the part that contains the questionable line; that is, you could enter RUN 80 and save yourself the time required for the first part of the program to run. Again, for a relatively short program as in the above example, this technique may not seem to be worth the trouble. But for long programs, which may take as much as several minutes to run, this variation of RUN may be a significant time saver.

The REM Statement — Keeping Order Within Your Programs

In the first part of this chapter, you learned how to use some special features of your Model 100 that facilitate programming. You learned how to use most of the special keys and how to edit programs in the EDIT Mode. You also learned how to use modifiers to make the LIST and RUN commands more versatile and help you write and debug long programs. In this section we'll introduce the REM statement, which can be used *within* a program to help keep order within the program and clarify how the program works.

All the BASIC statements we've introduced *do* something, like printing a number with PRINT. The REM (for "REMark") statement, however, simply gives you the chance to present some information for the benefit of the programmer (or anyone else who reads the listing of the program).

When to Use REM

It is a good idea to use REM whenever you need to clarify something in your program. One such clarification is a title and description of what the program does. For example, consider our program that prints a list of numbers and their squares. We may wish to insert the following REM statements to provide a title and some explanation:

```
1 REM---NAME:"SQUARE"-----
2 REM
3 REM---Program prints a list of
4 REM---numbers and their squares
5 REM
10 PRINT 1, 1*1
20 PRINT 2, 2*2
*
*
*
```

The program starting with line 10 is the one we used earlier. The REM statement in line 1 gives the name of the program, which could be anything we like. Here we've limited ourselves to a six-letter name so that if we want to save our program in memory (or on cassette tape, as described in the next chapter), we'll have a program name that can also function as the *filename* — the name of a stored program. As we'll see later, filenames must obey certain rules, one of which is that they cannot be longer than six letters.

The REM statements in lines 2 and 5, which have nothing written after them, serve purely as spacers that separate different parts of the program. The use of blank REM statements is particularly helpful in visually organizing longer programs. Lines 3 and 4 explain what the program does. It is sometimes a good idea to also include the author's name and the date — or, for that matter, any information that seems relevant to a person trying to understand the program.

We can abbreviate the REM statement by using an apostrophe in its place; for example, we can replace the first few lines in our program with the following equivalent statements:

```
1 '---NAME: "SQUARE"-----
2 '
3 '---Program prints a list of
4 '---numbers and their squares
5 '
*
*
*
```

The apostrophe is sometimes preferable to REM because it takes up less space. Also, you can use the apostrophe to comment at the *end* of a BASIC statement, as in this example:

```
50 PRINT 5/6 'a quotient
```

In this manner an apostrophe can sometimes be used effectively on the same line as a BASIC statement in order to explain the statement.

To summarize, it is a good idea to use REM statements throughout a program — especially a long and complex one — to identify the title and purpose of the program as well as to explain and clarify its inner workings. Later in this book you'll find many program examples that further illustrate the use of the REM statement.

Using Your Printer

With a printer you can obtain hard-copy program listings and program output. As useful as a printer is, though, it is neither essential to programming in BASIC nor required in the use of this book. We include this section for the benefit of those people who have access to a printer; if you don't have one, you may want to skip this section (if you read it, however, you might find yourself heading to your local Radio Shack Store to pick one up).

One way to use a printer is to print out a program listing. If you're still working on a program, listing it on paper gives you an overview that the screen, which is limited to eight lines, may not provide. You can also write notes on your printed page. If you're finished writing a program, it is often convenient to have a printed copy of it as a reference. A printer can also be

used to produce hard-copy program output, which is especially useful in providing a reference for various applications. Printed output can be as long as you like; it is not limited to the eight lines that fit onto your screen.

We'll now describe four printer-related commands and statements that enable you to print the two types of printed records discussed above.

The **PRINT** Key

Earlier in this chapter we briefly mentioned the command key labeled **PRINT**, which causes your printer to print whatever appears on your screen. To try it, clear the screen and enter the following program line:

```
10 PRINT 453 * .23
```

Then list and run this one-line program. When you're done, your screen will show the following:



Now press the **PRINT** key. If your printer is ready to print (properly connected and turned on), it will immediately start printing. The printed output will be identical to the screen image shown above. Well, almost identical — the only missing item on the printed page is the cursor, which the printer never prints. As you can see, you can use the **PRINT** key to get a printed program listing as well as printed program output. The **PRINT** key causes your printer to print *everything* on your screen (except the cursor).

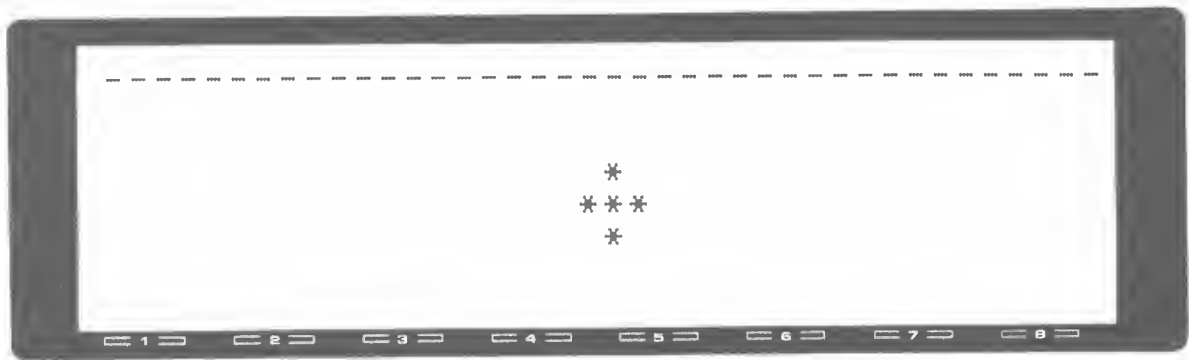
The LCOPY Statement

You can also produce hard-copy printed output of exactly what is on your screen by using the LCOPY instruction. Although LCOPY can be used as a direct command (you type LCOPY and press **ENTER**), its principal use is as a BASIC statement — that is, as part of a BASIC program. Let's try it on a program similar to the one we presented as the solution to the second exercise problem in Chapter 3: the only change we'll make is to *insert*

two new lines, lines 15 and 45. The purpose of line 15 is to mark the top of the screen; line 45 is our new LCOPY statement. Here is the modified program:

```
10 CSL
15 PRINT "-----"
   "
20 PRINT @139, "***"
30 PRINT @100, "*"
40 PRINT @180, "*"
45 LCOPY
50 END
```

Now run this program (remember, you can simply press **F4**). The output on your screen will be a line at the top and a small cross of asterisks in the center, like this:



Immediately after the screen image is completed, your printer will print out exactly what your screen shows. That's what LCOPY does — it directs your Model 100 to print out on a printer exactly what is on the screen at that time. Its effect is almost identical to running the above program without the LCOPY statement in line 45 and then pressing the **PRINT** key. The difference is that when LCOPY is executed, the BASIC prompt hasn't yet appeared on the screen, so it won't be printed on your printer, whereas the **PRINT** key will always cause the Ok prompt to show on the printed page.

LCOPY is the statement to use whenever you want to produce a printed copy of a screen image under *program control*.

The LLIST Command — Getting a Hard-Copy Program Listing

The LLIST command is designed to produce hard-copy program listings. Assuming you haven't erased the previous program, enter the direct

command LLIST. Your printer will print out the complete program, regardless of what appears on your screen. It's just like the LIST command except that your program is listed on the printer instead of the screen.

The advantage of LLIST is that it gives you a permanent record of your program. It is particularly useful in listing a program longer than the six lines that can be displayed on your screen at one time (your screen has eight lines, but one is occupied by the prompt, and another by the cursor). LLIST prints out your program no matter how long it is. Also, program lines printed with LLIST are not limited to forty characters the way your screen is. If you have long program lines that "fold over" on your screen, these lines will unfold when printed with LLIST. The maximum line length is determined by the printer, most of which have a width of eighty characters.

LLIST can also be appended in various ways to specify exactly which lines are to be printed out. These options, which are identical to those available with LIST, are summarized in Table 4-3.

Program Listings in This Book

Most BASIC program lines shown in this book will be less than forty characters wide, and their appearance will be identical to that on your Model 100 screen. Occasionally, however, program lines will need to be longer than forty characters. On your Model 100 screen, such program lines are folded over to the next line (it's even possible to write program lines more than five lines long!). Because it's a bit awkward to read such folded statements, from now on we will list these long program lines as they would be printed out on your printer with LLIST — that is, without any folding. We'll show a sixty-character program line as a single, sixty-character line, rather than as two separate lines of thirty-nine and twenty-one characters.

Command	Function
LLIST	Lists entire program
LLIST 50 – 120	Lists lines 50 through 120
LLIST – 350	Lists lines from beginning of program to line 350
LLIST 45 –	Lists lines beginning with 45 to end of program

Table 4-3. Options of LLIST

The LPRINT Statement

The last printer-related instruction we'll consider is LPRINT. LPRINT is identical to the PRINT statement except that it writes to your printer, whereas PRINT writes to your screen. Consider the following program:

```
10 PRINT "Good morning"
20 LPRINT "Good morning"
30 PRINT "3 * 15 ="; 3 * 15
40 LPRINT "3 * 15 ="; 3 * 15
50 END
```

When you run this program, lines 10 and 30 direct the output to the screen, and lines 20 and 40 direct the output to the printer. Because we asked the PRINT and LPRINT statements to print exactly the same thing, the output of this program on both the screen and the printer will be the following:

```
Good morning
3 * 15 = 45
```

As you can see, LPRINT can print both strings and numbers as well as the results of arithmetic operations.

Notice that we can obtain printed output similar to that obtained above by replacing the LPRINT statement in line 20 with LCOPY. That is, line 10 would write "Good morning" to the screen, and LCOPY would cause the printer to print whatever is on the screen, which includes the desired program output. The difference, though, is that LPRINT is more discriminating: LPRINT causes only the expression that *immediately follows* it to be printed (by the printer), whereas LCOPY prints everything that happens to be on the screen at that time.

The PRINT key	Causes printer to print a screen image
LCOPY	Has same effect as PRINT key, except that LCOPY can be used within a program to print a screen image
LLIST	Is a direct command that prints the program listing on the printer
LPRINT	Is identical to PRINT, except that LPRINT directs output to the printer instead of the screen

Table 4-4. Summary of BASIC instructions related to your printer

LPRINT is particularly useful if your program output is longer than eight lines. Suppose you want to write a program that produces a table of ten numbers and their squares (a number multiplied by itself). If you use PRINT to direct the output to the screen, you can see only eight lines at once (although you can use **PAUSE** to select which lines you look at). However, if LPRINT is used to direct the output to your *printer*, there will be no such limitation — your printed columns can be as long as you wish.

Summary

In this chapter we have been primarily concerned with how you can use your Model 100 to make programming easier. We described many of the special keys that enable you to give commands with a single keystroke. The first eight special keys are called programmable function keys, and they can be programmed to perform any simple direct command. Six of these keys have been preprogrammed at the factory, although you can reprogram them to suit your individual needs. The second set of special keys is the command keys, of which we described the **PRINT** key and the **PAUSE/BREAK** key.

Another extremely useful programming aid is the BASIC editor called EDIT, which is the built-in program that allows you to change a BASIC program with much greater ease than is possible in the BASIC Command Mode. Within EDIT, we can move the cursor to any part of the program and make insertions and two types of deletions using the **DEL/BKSP** key. You can also join different lines and break a single line into two.

We introduced variations of two familiar BASIC instructions that facilitate programming. The LIST command can be appended with numbers that specify exactly what parts of your program are to be listed. The RUN command can also be appended with a number that specifies the first program line to be executed. We also introduced the REM statement, which is used to write remarks to the programmer (or anyone else who reads and tries to understand your program).

For the benefit for those readers who have a printer available for their Model 100, we introduced four printer-related instructions: the **PRINT** key, LCOPY, LLIST, and LPRINT. These instructions give you great flexibility in printing program listings as well as program output.

Exercises

1. Program the programmable function keys (F6) and (F7) so that pressing (F6) clears your screen and pressing (F7) invokes the EDIT Mode.
2. Use EDIT to make some changes in the list of items and prices in the program "MY SHOPPING LIST" presented as a solution to an exercise in Chapter 3. The exact nature of the changes is not important; the idea is simply to practice making some changes.
3. Write a program that converts the temperatures 60, 62, 64, 66, 68, and 70 degrees Fahrenheit into degrees Centigrade. Use REM statements to write a title and a program explanation. Program output should be to the printer. Use the fact that to find the temperature in degrees Centigrade, you subtract 32 from the temperature in degrees Fahrenheit and multiply by 5/9.

Solutions

1. In the BASIC Command Mode, enter the following commands to reprogram the (F6) and (F7) keys:

```
Key 6, "CLS" + CHR$(13)
Key 7, "EDIT" + CHR$(13)
OK
```

We find the (F7) key particularly useful as a time saver when doing a lot of program editing.

2. We'll let you play (or struggle!) with that one.
3. There are many ways to do the required job. Here is our program:

```
10 REM---NAME: "FTOC"-----
20 REM
30 REM---Program Prints list of temps,
40 REM---from 60 to 70 deg, in deg, F
50 REM---and corresponding temp, in
60 REM---deg, C
70 REM
80 REM---MAIN PROGRAM-----
90 REM
100 LPRINT "Deg, F", "Deg, C"
110 LPRINT "-----", "-----"
120 LPRINT
```

```
130 LPRINT 60, (60 - 32)*5/9
140 LPRINT 62, (62 - 32)*5/9
150 LPRINT 64, (64 - 32)*5/9
160 LPRINT 66, (66 - 32)*5/9
170 LPRINT 68, (68 - 32)*5/9
180 LPRINT 70, (70 - 32)*5/9
190 END
```

Notice that we needed to use the LPRINT statements to get a printed copy of the *complete* output. Had we used PRINT in place of LPRINT and then added an LCOPY statement, we'd get only the last six lines of the output printed (by the printer).

5

Storing Your Program

Concepts

- Files and filenames
- Saving and loading programs in RAM memory
- Listing RAM files
- Modifying RAM files
- Erasing RAM files
- Saving and loading programs on cassette tape
- Listing cassette files
- Saving and loading programs on disk

Instructions

SAVE, LOAD, FILES, NAME...AS, KILL, CSAVE, CLOAD, LFILES

It takes time and effort to write good BASIC programs, and few of us would want to invest that time and effort if we could not save programs for future use. In this chapter we explain how you can store and retrieve your programs. We'll consider three different media for storing programs. The first is the memory in your Model 100, called RAM (for Random Access Memory) and the second is *cassette tape*, for which a cassette tape recorder/player is required. The third is the floppy disk, for which you'll need a disk drive. Cassette or disk storage is not essential to learning BASIC or using this book, so you needn't be concerned if you don't have a cassette recorder or disk drive.

Storing Your Programs in RAM Files

To begin our discussion of storing your programs in the internal memory of your Model 100, let's write a short program.

```

10 REM---NAME: "TREE"-----
20 REM
30 CLS
40 PRINT @60, "*"
50 PRINT @99, "***"
60 PRINT @138, "*****"
70 PRINT @180, "I"
90 END

```

When this program is run, it draws a simple tree near the center of your screen that looks like this:



This tree-drawing program is what we'll call the *current program* — the program that can be listed with the LIST command or run with the RUN command. Another term that will become very useful later is *active memory* — that part of your computer's memory in which the current program is stored.

One thing you already know about the memory of the Model 100 is that if you turn it off and then turn it back on, your program will still be there. The Model 100 has a special set of batteries that keeps the memory operating even while the computer is turned off. So far, our only way to erase the current program is to enter the NEW command.

RAM Files

Though we needn't be concerned about losing our present program when we turn off the Model 100, we must still address the problem of what to do if we want to write a new program and save our current program for later use. The NEW command doesn't help here because it erases the current program; we want to *save* this program *and* make room for a new one. What we need is a way of "filing" away programs that we're not presently using but want to keep for later use. We need an electronic filing cabinet.

The filing system we'll discuss here is the internal memory of the Model 100, called RAM (for "Random Access Memory"). This memory is like a library: you can file information away and then retrieve it whenever you wish. (Another kind of memory in the Model 100 is called ROM, for "Read Only Memory". As the name suggests, information can only be read from this type of memory. Information stored in ROM was loaded by the manufacturer.)

A program stored in this electronic filing cabinet (RAM) is called a RAM program file. A file, in general, is any body of information stored under a name or title called the *filename*. If we think of a BASIC program as a body of information stored in a manila folder, then the filename corresponds to the label on this folder. All the names that appear on the main menu when you turn on the Model 100 are filenames.

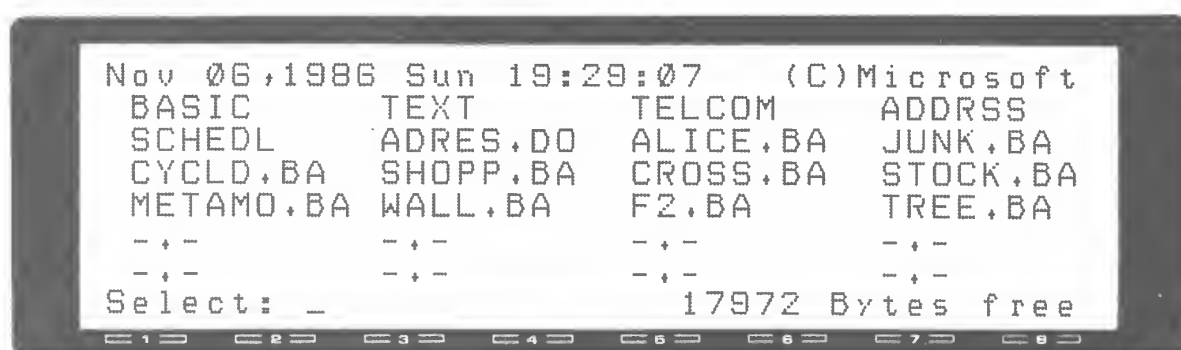
Saving a File in RAM

Let's save our tree-drawing program as a RAM file. Assuming you've entered this program into memory, enter the following command:

```
SAVE "TREE"
```

That's all there is to it; the current program is now stored as a RAM file under the filename TREE. The word *SAVE* instructs your Model 100 to save the current program under the filename specified right after the word *SAVE*. Notice the quotation marks that surround the filename. The first quotation mark is required; the second is optional, although we'll always include it for the sake of clarity.

How can we be sure that our program was actually saved as a RAM file? One way is to go back to the Main Menu by pressing the programmable function key (F8). Immediately the familiar menu appears on the screen, which looks something like the following:



As you know, the first five filenames beginning with BASIC and ending with SCHEDL are the names of programs built into the ROM of your Model 100. All the remaining names, which have periods in them, are names of RAM files. The files shown above are the ones that happened to be in our computer; your screen will undoubtedly show a different set of filenames. Notice that the file we saved looks very similar to the last file in the list. The filename we entered was TREE, whereas the menu listing has TREE.BA. The letters *BA* are called the *extension* of the filename and are added automatically when we save a file. This particular extension BA (for “BASIC”) tells us that TREE is the name of a BASIC or program file. (Notice that in our list of files above, most are BASIC files. Only one, called ADRES.DO, is a type of file called a *text* or *document* file, which we’ll discuss in a later chapter.)

The Model 100 automatically added the extension BA to the name TREE that we entered after the SAVE command. However, we could have included the extension ourselves when we originally saved our program; that is, we could have entered

```
SAVE "TREE.BA"
```

instead of SAVE “TREE”. The effect is the same: the program is stored under the name TREE.BA.

Figure 5-1 illustrates how you can think of the process of saving a current program, TREE.BA in our example, as a RAM BASIC file.

Now that we know our program has been stored as a RAM file, let’s return to the BASIC Command Mode (select BASIC from the main menu) and enter LIST to see whether the program we saved is still the current program. Nothing is listed, so the answer is “no” — there is no current program. That is, when we originally saved our program TREE as a RAM file and then switched to the main menu, the Model 100 “forgot” the current program as if we had entered NEW. That’s understandable, because the computer thought, “Well, the program is now saved in a RAM file, so it’s okay for me to forget the program as a current program; in fact, it’s a good idea to erase the current program because the user probably wants to start working with a new program.” Merely *saving* a program, however, doesn’t automatically erase the current one. This happens only when we enter the main menu, as we have done, or if we enter NEW. If we haven’t done either of these things, the current program won’t be erased.

Rules for Naming Files

Before we show you how to retrieve or load the RAM file TREE.BA back into active memory, we need to specify the rules for naming files.

Looking at the list of RAM files listed in the main menu should give you a pretty good idea of what these rules are. The first part of the filename must be made up of one to six characters, the first of which must be a letter. Our entry, TREE, clearly fits the bill. The second part of the filename, after the period, is the extension, which always consists of two letters. As mentioned

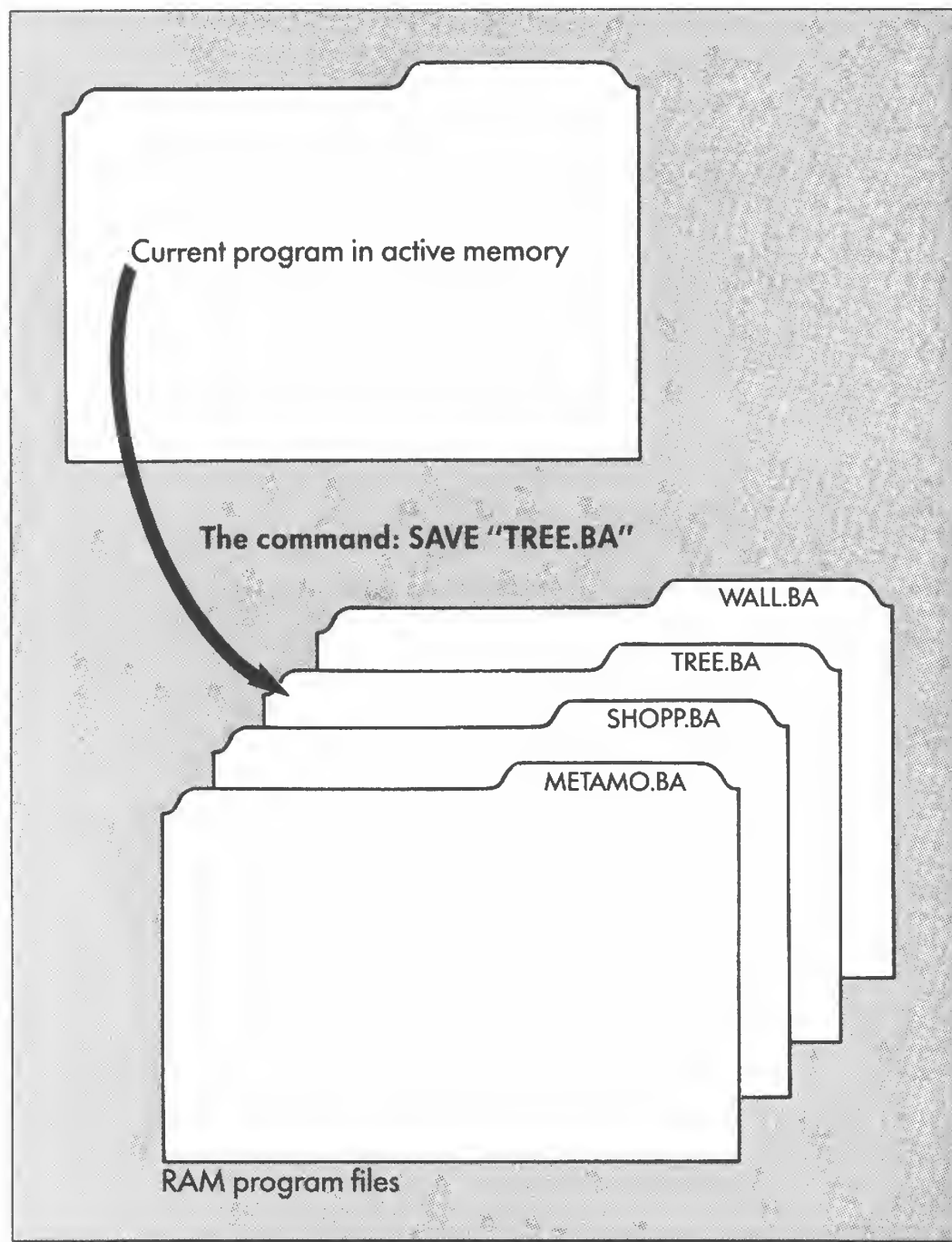


Figure 5-1. Saving a current program as a RAM file called TREE.BA

above, although the extension is optional when saving a file, it is always listed with the extension in the main menu. Also, as with any BASIC instruction, filenames may be entered in lowercase letters but will always be listed in uppercase letters.

The following table shows examples of filenames as entered using the SAVE command and as listed in the main menu:

Filename as Entered	Filename as Listed in Main Menu
SLOP	SLOP.BA
M6	M6.BA
M6.ba	M6.BA
tree1	TREE1.BA
catnip	CATNIP.BA

Specifying the Device

There is another option we can add for clarity, which is called the *device specification*. The device in which we are now storing files is the internal memory of the Model 100, RAM. The word RAM (followed by a colon) can be used as a prefix to our filename, to give "RAM:TREE.BA". So instead of saving our program by entering SAVE "TREE", we could have saved it this way:

```
SAVE "RAM:TREE.BA"
```

The complete phrase between the quotation marks - the filename prefixed by the device specification — is sometimes called the file specification. Why complicate our entry with an optional device specification? At this stage in learning about files, there really is no advantage. However, the device specification suggests that we can store information in devices other than RAM, such as the cassette recorder and the disk drive. Later in this chapter we'll use the device specification to store a current program on cassette tape and on disk. (The device specification further suggests a general format, or recipe, for sending any kind of information to any one of many devices that are part of, or may be attached to, the Model 100. We'll discuss this general technique of sending information in a later chapter on data files.)

Filename and Saving a BASIC Program

The command

SAVE "RAM: TREE, BA"

Optional extension consisting of two letters

First part of filename — up to six characters, the first of which must be a letter

Optional device specification

saves the current BASIC program under the filename TREE.BA in the internal memory (RAM) of the Model 100.

We can press the programmable function key **F3** in place of typing the word *SAVE*.

Loading a RAM File into Active Memory

What we've done so far is to save our tree-drawing program in a RAM program file called TREE.BA. We also entered the main menu to see if our file was indeed listed. In the process, our program was "erased" from active memory. Suppose we now want to run our tree-drawing program once again or make some changes in it; that is, we want to load the RAM program file TREE.BA back into active memory. There are two ways to do this: we can load it from within BASIC, or we can load it from the main menu.

Using the LOAD Command from BASIC

One way to retrieve our program as a current program is to use the BASIC command **LOAD**. To use it, get into BASIC and enter the following:

```
LOAD "TREE"
```

Notice that we omitted the extension BA. However, we could have included the extension; then the command would read

```
LOAD "TREE,BA"
```

When the BASIC prompt `Ok` reappears on the screen, the file `TREE.BA` should have been “loaded” as the current program. As you can see, the `LOAD` command has a syntax (the grammar or structure of an instruction) very similar to the `SAVE` command: the first word tells the Model 100 what to do — `LOAD` or `SAVE` — and the second word within the quotation marks is the name of the file to be loaded or saved. As with the `SAVE` command, we could also omit the quotation mark after the filename, but we’ll always include it for the sake of clarity.

If you want to load a program in order to run it, you would use the `LOAD` command as just described and then enter `RUN`. However, you can combine the sequence of commands `LOAD` and `RUN` by appending the `LOAD` command with an `R`. To try it out, enter `NEW` to erase the current program and then enter the following command:

```
LOAD "TREE.BA",R
```

After a brief moment, the tree appears on your screen! Not only has the file `TREE.BA` been loaded as a current program, but this program has also executed — as if you had also entered `RUN`. Using this appended version of the `LOAD` command offers no significant advantage over first entering `LOAD` and then entering `RUN`, especially because we can use the `(F4)` key in place of entering `RUN`. However, when `LOAD` is used as a BASIC *statement* (virtually all the BASIC instructions we’ve used as commands can also be used as *statements*), the `R` option of `LOAD` becomes more valuable. For example, the statement

```
50 LOAD "TREE.BA",R
```

which may be part of a more complete program, causes the file `TREE.BA` to be loaded into active memory and run. That is, the `LOAD` statement, which can be part of one program, can cause another program to be loaded and run. Using this technique is one way to link or chain different programs together. Programs are usually chained only in rather advanced applications, however, so we won’t explore this topic any further here.

The LOAD Command

The command

LOAD "TREE.BA",R

Filename with extension

Causes loaded program to be run; optional

causes the RAM file TREE.BA to be loaded into active memory and immediately run. Use of both the filename extension and R are optional. The device specification RAM may also be included as a prefix to the filename, as in "RAM:TREE.BA".

Loading a RAM File from the Main Menu

The second way to load a program RAM file into current memory involves three simple steps. First, return to the main menu by pressing key **(F8)**. Then use the cursor control keys (as described in Chapter 4) to move the cursor (the main menu cursor is much longer than the BASIC cursor) to coincide with the RAM file you want to load into active memory — TREE.BA, in our example. You can guess the third step: press the **(ENTER)** key and, after a brief moment — voila! — our tree appears on the screen! Selecting a RAM file from the main menu loads the RAM file into active memory *and* runs the program. The result is the same as entering the BASIC command LOAD "TREE.BA",R.

Listing Your Files from BASIC

When saving and loading RAM files, it is often helpful to look at our file inventory to see exactly what files have been saved. You already know one way to list all your RAM files: simply press the **(F8)** key to return to the main menu. Another method, however, allows you to list your RAM files without having to leave BASIC. To try it, enter the following command:

FILES

The result is a screen display something like this:



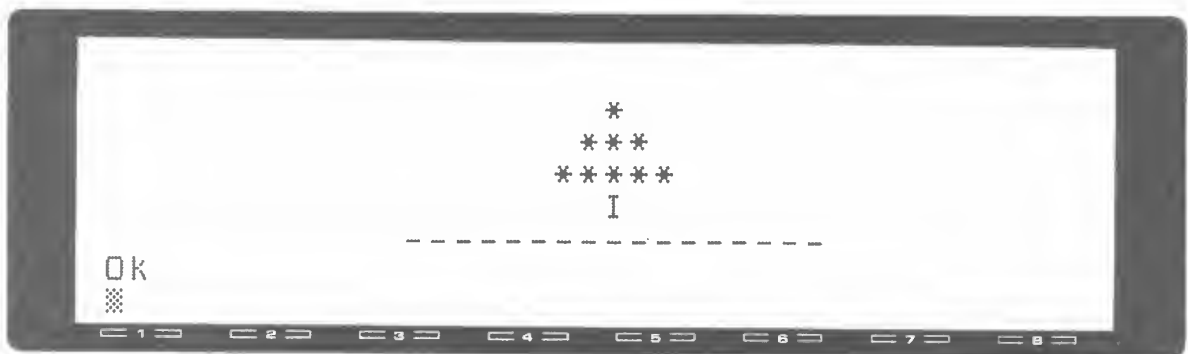
Perhaps you'll recognize these as the same RAM files that appeared on our main menu. A minor difference in the way that FILES and the main menu lists the files is that FILES inserts spaces between the first part of the filename and the extension so that all the periods line up in neat columns. Another difference is that FILES doesn't list the five built-in programs BASIC through SCHEDL. Using the FILES command is the most convenient way to list RAM files when you are working in BASIC.

Modifying a RAM File

You now know how to save a program as a RAM file, how to list all your RAM files, and how to load a RAM file into current memory so that you can again run your program. If you are like most programmers, though, you'll often want to modify an existing RAM program file. For example, suppose you want to add a horizontal line under the tree in our RAM file TREE.BA to suggest the ground on which it stands. Load this file into active memory using the LOAD command. (We've already done this in the previous section, but check to make sure that you have the program TREE.BA in active memory by entering LIST.) Then make the required change in the current program. One way to modify this program so that it draws a horizontal line below the tree is to add the following statement:

```
80 PRINT @212, "-----"
```


Now run the modified program to be sure that it does what you want. This is what the tree now looks like:



How can you save this modification in your RAM file TREE.BA? A reasonable approach would be to use the SAVE command. Let's try it:

```
SAVE "TREE.BA"  
?FC Error  
OK
```

An error message! This particular error message is an abbreviation for “Illegal Function Call”, which means that you can't use the SAVE command the way you intended. The problem is that you already have a RAM file called TREE.BA, and BASIC on your Model 100 won't let you “write over” an existing file. There are, however, two ways to store the modified file. Both methods are easy to use, although it's not obvious that they should work. The first is to enter NEW. Yes, NEW. Don't worry about losing your program: you will lose it as a *current* program, but before it is erased from active memory, your Model 100 will automatically store it as a RAM file under the original filename TREE.BA. By entering NEW, you save your *modified* program under the filename of the original RAM file that you loaded into active memory. To make sure that what we're telling you is correct, enter the command LOAD “TREE” once again and then run the current program — you'll see that this program is the *modified* version, which draws a horizontal “ground” line in addition to the original tree.

Another way to save modifications made on a program loaded from a RAM file is to return to the main menu by pressing the (F8) key. If you then return to BASIC and try to list the program you just modified, you'll find that it's been erased from active memory — but saved as a RAM file under the original filename. Returning to the main menu and reentering BASIC has the same effect as entering NEW in BASIC. Remember, however, that we're talking only about this very special situation in which we loaded a RAM file into active memory. If you're dealing with a new program

that you've just written from scratch rather than loaded from a RAM file, entering the main menu and returning to BASIC will *not* erase the current program and will not save your program in a RAM file. Likewise, entering NEW *will* erase your current program without automatically saving it in RAM.

Saving Two Copies of a BASIC Program

In the last section we discovered that although we could load the RAM file TREE.BA into active memory and then make a modification, we could not save the modified version under a different filename. Even if we had not made any changes in the original program TREE.BA, we would still find it impossible to make a second copy of the *same* BASIC program under a *different* filename. The explanation for this puzzling limitation has to do with the way the Model 100 memory works: when you load a program from the main menu into active memory, it is not really moved from a RAM file to active memory; instead, a *pointer* is moved to it that says, "Now you're a current program, not a RAM file"; the program itself remains in the same storage location. Conversely, when you write a new program in active memory and then save it as a RAM file, it is not really moved from active memory to a RAM file; a pointer simply identifies the original program as a RAM file rather than a current program. So although it may appear that there are two copies of a program — one in active memory and one in a RAM file (the name of which appears in the main menu), — there is actually only one copy, which can have only one filename.

Is there *any* way to save a program under two different filenames? Such a procedure would be especially valuable in generating different versions of basically the same program. For example, when developing complex programs, it would be helpful to have a copy of the "old" version in case the new version wasn't satisfactory and you wanted to return to the earlier one.

There *is* a way to store a BASIC program under two different filenames, but it involves several steps and an intermediate file. The following is a recipe for making duplicate copies of the same BASIC program (it's not important now that you understand *why* this method works, so don't be concerned if this procedure seems a bit mysterious). Suppose you want to make two copies of the tree-drawing program already stored as a RAM file under the filename TREE.BA. First, load TREE.BA into active memory by getting into BASIC and using the LOAD command (you can also select the TREE.BA directly from the main menu). To check that the program is now current, you can enter RUN; a treelike figure should appear on your screen.

The next step uses a new version of the familiar SAVE command. Enter the following:

```
SAVE "TREE",A
```

The new feature of this command is the comma followed by the letter A, which saves the current program in a special code called ASCII (hence the letter A at the end of the SAVE command). You can see this if you now press **(F8)** to return to the main menu: you'll find the original file TREE.BA as well as a new file called TREE.DO. The latter is one of those document or text files that we mentioned before; it is the same kind of file as those generated by the built-in program TEXT.

The next step is to load TREE.DO as a current BASIC program using the familiar command

```
LOAD "TREE.DO"
```

During the brief pause while the document file is converted to a BASIC program file, the screen will flash the message "WAIT". When the BASIC prompt appears, you can now use the SAVE command in the way we used before to save this current program under a *new* filename. For example, enter the following command:

```
SAVE "TREE2"
```

To see what you've done, press **(F8)** to return to the main menu. You should now have *two* BASIC RAM files called TREE.BA (the original) and TREE2.BA (the copy). In addition, you'll have the document file TREE.DO, which is no longer needed. (You can erase it using a procedure described later in this chapter.) The significant result of the whole process just described is that you now have two copies of the same BASIC program under different filenames, and you are now free to make changes in either copy without affecting the other. Figure 5-2 summarizes the process of creating two copies of a BASIC program under different filenames.

Saving Time with the Programmable Function Keys

In Chapter 4 we mentioned several programmable function keys without fully explaining their function. We are now ready to use these keys to help us enter the commands introduced in this chapter.

To see what the relevant programmable function keys are, press the **(LABEL)** key. The first three keys are identified as follows:

(F1) = FILE

(F2) = LOAD

(F3) = SAVE

The word *FILE* refers to the command FILES. Instead of typing the word *FILES* and pressing **(ENTER)**, you can simply press the **(F1)** key! Likewise, instead of entering LOAD "TREE", you can press **(F2)**; your screen will immediately show the following:

```
LOAD "█
```

Now the only thing left for you to do is fill in the filename and press **(ENTER)**. Similarly, pressing **(F3)** causes your Model 100 to print out the first part of the SAVE command, as shown below:

```
SAVE "█
```

Again, to finish the command, simply enter the filename.

These preprogrammed function keys are a great convenience when listing your files and when loading and saving your programs as files.

Renaming a RAM File

It is often useful to change a filename (we're not talking about duplicating a program under different filenames, as discussed earlier, but simply about changing the name of an existing program file). Suppose you want to change the name of the file TREE.BA to PINE.BA. Assuming you're in the

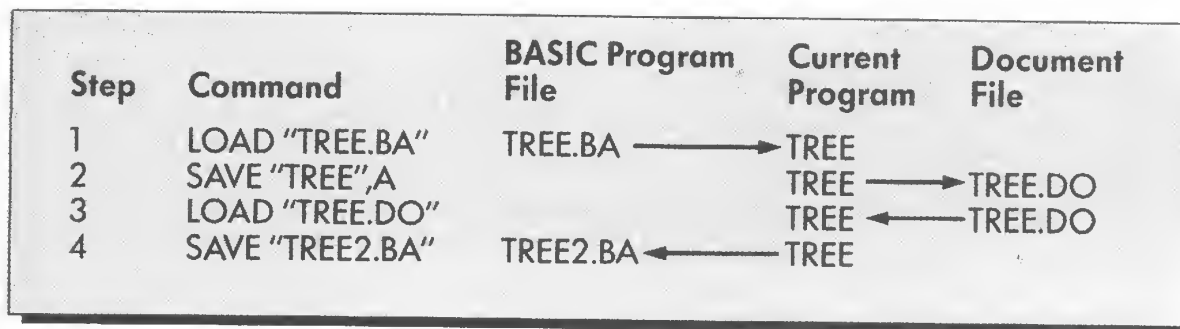


Figure 5-2. Creating a copy of a BASIC program RAM file under a different filename

BASIC Command Mode, use the NAME...AS command in the following manner:

```
NAME "TREE.BA" AS "PINE.BA"  
OK
```

Now use the FILES command to list the current RAM files. You'll see that the filename PINE.BA has replaced TREE.BA. Notice that we've used the filename extensions in the NAME...AS command. In contrast to using SAVE and LOAD, the filename extension must be used when renaming files.

Erasing a RAM File

Your Model 100 can store up to nineteen RAM files. You won't find it at all difficult to use up your whole file allotment in a very short time! Clearly, you'll need a way of erasing or "weeding out" unwanted files. The KILL command is designed for this task. Although this command is easy to use, there are a few potential pitfalls you'll need to be aware of.

To learn how KILL is used, create a short nonsense program that you won't mind erasing and save it as a RAM file called JUNK.BA. If you now try to use the KILL command to erase the file, you'll find that you can't do it — you'll get an error message! Before you can erase the JUNK.BA file you must do one of two things: either enter NEW or return to the main menu and then go back to BASIC. Now you're ready to enter the KILL command.

```
KILL "JUNK.BA"
```

If you now check your RAM file inventory (enter FILES), you'll see that the file is gone. The KILL command, like NAME...AS, requires the use of the filename *with extension*.

Any kind of file can be KILLED; it doesn't have to be a BASIC program. For example, the command KILL "LETTER.DO" will erase the document file LETTER.

You've learned how to save, load, list, change, erase, and rename files. You now have a convenient and practical means to store your programs in the relatively permanent internal memory of your Model 100. The following box summarizes what you've learned about storing programs as RAM files:

Manipulating RAM Files

Command	Effect
SAVE "TREE.BA"	Saves current program in RAM file under name TREE.BA (extension .BA is optional)
LOAD "TREE.BA"	Loads RAM file TREE.BA into active memory (extension .BA is optional)
FILES	Lists all RAM files
NAME "TREE.BA" AS "PINE.BA"	Renames file TREE.BA as PINE.BA
KILL "GORP.BA"	Erases the file GORP.BA

Cassette Files

RAM files are convenient and easy to use, but because their number is limited to nineteen, you may need additional file storage space in another storage medium. Also, you may want more permanent storage than a battery-operated RAM memory can provide. Although your Model 100 has two sets of batteries and an excellent "low-battery" warning system, you can lose all your RAM files if your Model 100 is left unattended for a long period of time without being plugged into a powerpack.

A cassette recorder/player can take care of both these problems. Instead of (or in addition to) storing programs in RAM as RAM files, we can store programs on cassette tape as *cassette files*. The commands for dealing with cassette files are quite similar to those related to RAM files, so much of what you've learned about RAM files still applies. However, because you'll be learning how to deal with a new piece of equipment — the recorder/player — it's important that you *practice* saving and loading cassette files before you actually work with valuable files you can't afford to lose!

Getting the Recorder/Player Ready

You'll need to acquire an appropriate cassette recorder/player and a special cassette cartridge with leaderless magnetic tape (magnetic tape that doesn't have the usual nonmagnetic blank "leader" at the beginning and end). Connect the recorder/player to your Model 100 as explained in your Model 100 manual. Assuming your leaderless cassette cartridge is brand new, insert it into the recorder/player, make sure it's completely rewound, set the counter to zero, and you're ready to go.

Saving a Cassette File

Let's tackle the problem of saving the RAM file PINE.BA (which we saved to RAM in the first part of this chapter) as a CAS (for "cassette") file. If PINE.BA is not already your current program, load it from the RAM file into active memory. Only programs in active memory can be saved in CAS files.

Because we are going to save a file on cassette tape — basically a recording process — we need to set the recorder/player to *Record*. This usually involves pressing the Play and Record buttons simultaneously. Notice that the cassette reels don't start to turn, as they would if the recorder/player weren't hooked up to your Model 100 (if they do turn, check your connections).

Now you're ready to give instructions to the Model 100. To save your current program as a CAS file, enter the following command:

```
SAVE "CAS:PINE"  
OK
```

Your cassette reels will turn for a short while. When they stop, the BASIC Ok prompt will reappear to indicate that the job is done — the current program has been saved as a cassette file under the name PINE.

CAS is the device specification in the SAVE command that tells your Model 100 to save the current program in a cassette file. The difference between saving RAM and CAS files is that in creating a RAM file the device specification is optional, whereas in saving a CAS file it is mandatory.

The rules for naming cassette files are identical to those for naming RAM files, which means that the filename can be up to six characters long, the first of which must be a letter. The extension is optional, as it is in saving and loading RAM files.

Another way to save a current program in a CAS file is by means of the CSAVE command. Try it with the same tree-drawing program you already have as a current program. Enter the following command:

```
CSAVE "PINE"
```

Your system will behave as before. You now have two CAS files named PINE — that's all right for cassette files (but not for RAM files). Of course, we could have used any legal filename. Notice that we didn't use a device specification this time: using CSAVE instead of SAVE tells your Model 100 that you are saving a program on cassette tape rather than in RAM. Again, you don't need to use the file extension.

Saving a Program as a Cassette File

Either of the following two commands will write a current BASIC program into a cassette file by the name of TREE:

```
SAVE "CAS:TREE"  
CSAVE "TREE"
```

The same rules apply for naming cassette files as for naming RAM files.

Loading a Cassette File into Current Memory

The whole point of saving a program as a cassette file is to be able to retrieve it later, that is, load it into active memory. You can probably already guess the appropriate command, but before you tell your Model 100 what to do, you need to prepare the recorder/player.

Preparing the Recorder/Player

Your Model 100 cannot rewind the tape, so you must do it yourself. First, press the Stop button on the recorder/player to release it from the Record Mode; next, press the Rewind button to return the tape to its original starting point. On some tape recorder/players not specifically designed for use with your Model 100, before you can rewind the tape you must first disconnect the plug from your recorder at the location labeled Rem or Remote. The plug itself should be one of the two gray plugs — the one with the smaller shaft. By disconnecting this plug you put the recorder

under manual control so that you can rewind it. After rewinding the tape, reconnect the gray plug. Finally, you must set the recorder/player to Play Mode by pressing the Play button. The Play Mode enables the recorder to send information to the Model 100. At last you are ready to load your previously saved cassette file PINE. As you can see, it is easier to deal with RAM files than with cassette files!

The LOAD and CLOAD Commands

The hard part — preparing your cassette recorder — is done. Now you need only enter the following LOAD command:

```
LOAD "CAS:PINE"
```

The tape should start moving and your Model 100 will start to make some high-pitched noises, indicating that it is searching for a program under the filename PINE. When your computer has found the file PINE, it prints out the following message:

```
Found:PINE  
Ok
```

When the Ok appears, the cassette file PINE will be your current program. Again, except for the inclusion of the device specification CAS, this use of the LOAD command is identical to that used to load a program from a RAM file. The prefix CAS tells the Model 100 to look for a cassette file rather than for a RAM file.

An alternate way to load a cassette file into active memory is to use the CLOAD command. Rewind the tape and try the following command:

```
CLOAD "PINE"
```

Dealing with Many Cassette Files

Suppose that you've saved several files on a single cassette tape. How does the Model 100 find a particular file, for example, near the end of the tape? And how do we list the names of files on a cassette tape (if we've forgotten)? To answer these questions, save a few more files on your cassette tape — what programs you save and what you call them doesn't matter. For the purpose of the following discussion, we've saved two more files, SHOPP.BA and METAMO.BA, in addition to the two copies of PINE saved earlier.

Now let's try to load the file METAMO.BA, which is close to the end of our file list. One way to do it is to rewind the cassette tape, set the recorder to Play, and enter the command

```
LOAD "CAS:METAMO.BA"
```

As before, your Model 100 will make some noises indicating that it is searching for the particular file. Because this file is not the first file, it will take a little time for your computer and recorder to find it. Each time your computer skips over one of the stored files you are *not* looking for, it prints out a message like the following:

```
SKIP :PINE
```

By the time the file METAMO is found and loaded into current memory, the following will have appeared on your screen:

```
LOAD "RAM:METAMO.BA"  
SKIP :PINE  
SKIP :PINE  
SKIP :SHOPP  
Found:METAMO  
OK
```

Three programs had to be skipped before METAMO was found.

A time-saving technique is to preset the tape to the correct starting position by using the counter on your recorder. As mentioned before, the counter should be set to zero when the tape is completely rewound. Because you want to know the counter value near the beginning of each program, you must keep a record of the counter value just before you record each program. Then, to load a program, you need only set the tape at a position slightly preceding the counter value that identifies the beginning of the file.

Few of us are so well organized that we always know exactly what files are on our cassette tapes. Is there a way to list all the files stored on a cassette tape? We don't have an available command that is comparable to the FILES command for listing RAM files. However, it's not hard to trick the Model 100 into giving us a file inventory; we need only ask it to search for a file that we know doesn't exist. During the searching process, the Model 100 will print out all the files that it skipped in the search, that is, all the files on the cassette tape! Trying this method on our tape gives the following result:

```

CLOAD "X"           ← This file doesn't exist
SKIP :PINE
SKIP :PINE
SKIP :SHOPP
SKIP :METAMO
SKIP :JUNK

```

At this point it's up to us to realize when we've skipped the last file on the tape — the buzzing noise stops — and to press the **BREAK** key to stop the search.

Loading Without a Filename

Perhaps surprisingly, you can also use the LOAD and CLOAD commands without specifying the filename (something you can't do with RAM files). Two versions of the nameless load commands are:

```

LOAD "CAS:"
CLOAD

```

Both of these have the effect of loading the first file your recorder reads. If the tape is completely rewound when either of the above commands are entered, the first file on the tape will be loaded. If, however, you've already advanced the tape to the beginning of, say, the fourth program (by using Play or Fast Forward), the above commands will load the fourth program. On our tape, that's the file METAMO.

Loading a Cassette File into Active Memory

The cassette file OAK can be loaded into active memory by using either of the following commands:

```

LOAD "CAS:OAK"
CLOAD "OAK"

```

If the filename is omitted, the first file read by the recorder/player is loaded into active memory.

These commands can also be appended by an *R* to cause the loaded program to run immediately. Examples are:

```

LOAD "CAS:OAK",R
CLOAD "OAK",R

```

Disk Files

You've seen how to use RAM and cassette files to store BASIC programs. We can also store information by means of the *disk file*. A disk file is a BASIC program or text that is stored on a magnetic disk called a floppy disk ("floppy" because it is flexible) or simply *disk*. Disk files have the convenience of RAM files and the permanence of cassette files. A single floppy disk also has about five times the capacity of the maximum RAM storage available on the Model 100. Disk files, however, require the use of an additional piece of equipment called a *disk drive*. In this section we'll assume that you have a disk drive and that you have prepared your system to be ready to run disk-BASIC (see the *TRS-80® Model 100 Portable Computer* manual or the book *Introducing the TRS-80® Model 100* by Diane Burns and S. Venit [New York: Plume/Waite, New American Library, 1984]) The disk drive also gives you the option of using a TV set in place of the LCD display on the Model 100.

Saving and Listing Disk Files

Saving and loading BASIC programs as disk files is similar to saving and loading RAM and cassette files, so much of what you already know about files is applicable to disk files. Let's begin by saving the previous tree-drawing program as a disk file called SPRUCE. We'll assume that your disk drive is turned on and ready to be used to store BASIC programs on a formatted disk.

First, the program to be saved as a disk file must be in active memory; if you still have the RAM file PINE.BA or TREE.BA saved previously, simply load this file as a current program. Then enter the following command:

```
SAVE "0:SPRUCE.BA"
```

After a brief sound from your disk drive, the BASIC prompt reappears, indicating that your program has been saved as a disk file. The only difference between this and the SAVE commands we've used before is the device specification — the zero right before the colon. This zero tells your Model 100 that the current program is to be saved on the disk in the disk drive identified by the number "0". If you have two disk drives, you can also save a current program to a disk in the second drive by specifying the prefix "1". All the rules for naming RAM and cassette files apply to disk files. For example, the filename cannot exceed six characters, the first of which must be a letter; also, the extension BA is optional.

To check that your program has actually been saved onto disk, list all the files presently stored on your disk by using the following command:

```
LFILES 0
```

As before, the 0 specifies the number of the disk drive. (The files on a second disk drive can be listed by entering LFILES 1.) Assuming that the disk we're using is a copy of the Operating System Disk supplied by the manufacturer, your screen should now display something like the following:

```
LFILES
SYSTEM VER 01.00.00
FORMAT,      1      BACKUP,      2
BACKUP.SNG 2      SPRUCE.BA      1
153.00 K AVAILABLE
OK
```

Because the file SPRUCE appears in the listing, you can be confident that your program has indeed been saved as a disk file. The first three files listed are *system files* supplied by the manufacturer; they enable you to format new disks and to make backup copies of specific files, as well as a whole disk. The number following each listed file specifies the amount of storage space allocated to the file in units called *clusters*. A cluster contains 2.25 kilobytes of information. The phrase "153.00 K AVAILABLE" tells you how many kilobytes of memory space remain on your disk (an "empty" disk has 160 kilobytes of free memory).

As mentioned earlier, the extension BA is optional. If it is not used in the SAVE command, then LFILES would list the file as "SPRUCE." — that is, *without* the extension but *with* a period that identifies the file as a BASIC program. Note the difference in the way that RAM program files and disk program files are listed: RAM files are always listed (using FILES) with the BA extension, whereas disk files are listed (using LFILES) with the extension only if it is included in the SAVE command.

Loading a Disk File into Active Memory

Now that we have a disk file called SPRUCE.BA, let's load this file back into the active memory of the Model 100. Simply enter the following command:

```
LOAD "0:SPRUCE.BA"
```

After some sounds from the disk drive, the BASIC prompt reappears. SPRUCE.BA should now be the current program. Again, note the device specification "0" required whenever a disk file in the first disk drive is

specified — that's the main difference between loading RAM files and disk files. Another, more subtle difference is the following: if, as in our example, we originally saved the file using the extension BA so that the file is listed as SPRUCE.BA (using LFILES), we must also use the extension BA in the LOAD command. On the other hand, if we originally omitted the extension BA in saving the file so that it is listed as SPRUCE., we must also omit the extension in the LOAD command. To avoid difficulties arising from inconsistent use of extensions, use the command LFILES to check how a disk file is listed before loading it, or be consistent in how you specify filenames — that is, always include the BA extension or always omit it.

Erasing a Disk File

We can erase disk files in much the same way as we erase RAM and cassette files. To erase the disk file SPRUCE.BA, enter the following command:

```
KILL "0:SPRUCE.BA"
```

Now, using the LFILES command, you'll see that the file SPRUCE.BA has been erased. If you had originally saved your disk file as SPRUCE *without* the extension, you would also have to omit the extension in using the KILL command.

Using Disk Files

The following commands illustrate how to save, load, and erase a disk file and how to list all disk files:

SAVE "0:LASER.BA"	← Saves current program to disk file 'LASER'
↑	
Device specification: 0 for the first disk drive	Optional extension
1 for the second disk drive	
LOAD "0:LASER.BA"	← Loads disk file 'LASER' into active memory
KILL "0:LASER.BA"	← Erases disk file 'LASER'
LFILES 0	← Lists all disk files

The usage of the extension BA must be consistent: if a program has been saved with the extension, all subsequent commands relating to this file must also be used with the extension.

Summary

In this chapter we have dealt with storing BASIC programs as RAM files and as cassette files. A file is simply a program or other body of information that has been stored or saved under a given filename. The SAVE command can be used to save a current BASIC program as a RAM or cassette CAS file, or a disk (0 or 1) file. The LOAD command can be used to load a RAM, cassette, or disk file into active memory so that the program can be listed or run. Special commands for saving and loading cassette files are CSAVE and CLOAD.

RAM files are more convenient than cassette files to use. They are faster to save and to load, and we needn't worry about exactly where they are located within the RAM memory or in what order they were stored. Also, RAM files are more versatile because we can rename a RAM file with NAME...AS, and we can easily delete a RAM file with the KILL command. None of those conveniences are available to us when dealing with cassette files. On the other hand, cassette files can be a valuable supplement to RAM files: because the number of cassette files is related only to the length of a tape and the number of tapes we're willing to buy, cassette tape is a good medium for more permanent storage of our BASIC programs. Also, storage on cassette tape is "safer" than in RAM because magnetic tape doesn't require batteries to maintain its memory.

Disk files combine many of the advantages of both RAM and cassette files. Disk files are as easy to save, load, list, and erase as RAM files; disk file commands are also executed very rapidly — almost as rapidly as RAM files. On the other hand, disk storage has the permanence and storage capacity of cassette storage. Disk files are a tool that significantly extends the versatility and power of your Model 100.

Exercises

1. Write a short program and save it in RAM under the filename CAT1.BA. Then:
 - a. Return to the main menu and check to see if CAT1.BA is listed.
 - b. Return to BASIC and load and run CAT1.BA.
 - c. Create a RAM copy of the file CAT1.BA and call it CAT2.BA.

- d. Make some modifications in the file CAT2.BA.
- e. Erase the file CAT1.BA and rename CAT2.BA as CAT.BA.

2. Save the RAM file CAT.BA as a cassette file under the name MOUSER. List your cassette files to be sure that MOUSER is one of your files; then load it back into the active memory.

Solutions

1. Here's our program, although yours, of course, is probably different:

```

10 REM---NAME:"CAT1"-----
20 CLS
30 PRINT "      ***"
40 PRINT "      ***"
50 PRINT "     *****"
60 PRINT "    *      *"
70 PRINT "    *      *"
80 PRINT "   * * * *   *"
90 PRINT "   * * * *   *"
100 PRINT "   *****";
110 PRINT @0, " ";
120 END

```

Note the semicolons at the end of lines 100 and 110; their purpose is to suppress the carriage return otherwise executed after each PRINT statement. The suppression of the carriage returns is necessary to keep the picture from scrolling.

To save this program as a RAM file with the filename CAT1, enter the command SAVE "CAT1.BA". Remember, though, that the extension BA is optional.

To return to the main menu, press the **(F8)** key. CAT1.BA should be listed among all your files.

With the main menu cursor on the word *BASIC*, press **(ENTER)**. One way you can now load and run the program is to enter the following command:

```
LOAD "CAT1.BA",R
```

The BA extension is optional.

To generate a duplicate copy of CAT1.BA under a new name, say, CAT2.BA, enter the following instructions:

- a. Enter SAVE "CAT1",A and return to the main menu or enter NEW.
- b. In the BASIC Program Mode, enter the command LOAD "CAT1.DO".

c. Then enter the command SAVE "CAT2.BA". Now check the main menu to see if you indeed have the two BASIC program files CAT1.BA and CAT2.BA.

To modify CAT2.BA, load this program from the BASIC Command Mode and make some changes in the program by entering the EDIT Mode (described in Chapter 4). The modifications we made change the position of the cat's tail as shown below. Specifically, we modified lines 70-100 to look like the following:

```
70 PRINT "      *      *      *"
80 PRINT "      * * * *      *"
90 PRINT "      * * * *      *"
100 PRINT "      * * * *      *";
```

To erase the file CAT1, enter the command KILL "CAT1.BA". Note that the extension BA is required when using the KILL command. To rename the file CAT2 to CAT, enter the command NAME "CAT2.BA" AS "CAT.BA".

2. Load the RAM file CAT into active memory and prepare the recorder/player by positioning the tape to an appropriate starting point and setting the recorder to the Record Mode. Then enter the command SAVE "CAS:MOUSER" or CSAVE "MOUSER".

To list your cassette files, rewind the tape and enter the command LOAD "CAS:X" or CLOAD "X" (assuming that none of your files have the name X).

6

Introduction to Variables

Concepts

Variables

Variable names

Types of variables: numeric and string

Arithmetic operations with variables

Rules for evaluating arithmetic expressions

*I*n this chapter we discuss one of the most important and useful concepts in programming: the variable. All but the most simple programs contain variables, which allow us to write programs that give the computer general rather than specific instructions for performing certain tasks. You already know how to instruct the computer to use PRINT to display the product of 3 and 5, for example, but that product is limited to those specific numbers. It's important to be able to write a statement such as, "PRINT the product of number A and number B" — a statement that can be used for *any* numbers A and B. Variables allow us to write such general, flexible instructions. They are one of the tools that make the computer such a powerful device.

Numeric Variables

Let's start with the simple task of printing out the height of a room in feet. Let's imagine that we're dealing with a typical room that's eight feet high. Here's the program without variables:

```
10 PRINT 8
20 END
RUN
8
OK
```

There's another way to do the same thing, however. It's a little more involved, but there is a payoff that will become apparent later. Try this program:

```
10 HEIGHT = 8
20 PRINT HEIGHT
30 END
RUN
8
OK
```

The output is the same as that produced by the previous program — namely, 8. But this time we used the variable called “HEIGHT”. This is what happened: line 10 instructed the computer to make room in its memory for a quantity called HEIGHT (our variable) and to give it the particular *value* of 8. In computer talk, we'd say that the variable HEIGHT is “*assigned the value 8*”. Line 20 printed the value of the variable HEIGHT, not the name “HEIGHT” as you might have expected. Remember that HEIGHT in line 20 doesn't have any quotation marks around it, so it is not a string. When your computer encounters letters like HEIGHT that are not enclosed by quotation marks, it knows that it is looking at a variable.

To help further define the concept of a variable, add two more lines to the above program so that it looks like this:

```
10 HEIGHT = 8
20 PRINT HEIGHT
30 HEIGHT = 12
40 PRINT HEIGHT
50 END
RUN
8
12
OK
```

The point here is that there is nothing permanent about the particular value of a variable; it can easily be changed, as it is in line 30. The PRINT statement in line 40 then prints out the new or most recent value of the variable “HEIGHT”. The variable's *name* is permanent (or intrinsic), but its *value* is temporary.

Assigning Numeric Variables

Numeric constants can be assigned to variables by means of the equal sign in the following way:

Numeric variable name = numeric constant

The following examples illustrate BASIC statements that assign values to variables:

```
40 HEIGHT = 16
50 B = 34
60 PRICE = 3
```

Variable Boxes

A computer thinks about a variable in pretty much the same way we think about a house: a house is a fairly permanent box (we hope) with a permanent address. The address identifies the house in the same way that a variable name (such as HEIGHT) identifies the variable. The people living in the house are like the value of a variable: different people can move in and out of a particular house the way that different values can be assigned to the same variable.

To get a really clear sense of variables, you need to understand how the computer stores variables in its memory. A certain amount of memory is allocated for each variable *name*, and, associated with the name, a certain amount of memory is reserved for the *value* of the variable. So if you could look inside your computer with magic eyes, you'd see stacks of memory boxes, as shown here:

	*	
	*	
HEIGHT	16	Variable HEIGHT with assigned value 16
A	5	Variable A with assigned value 5
AMT	-546	Variable AMT with assigned value - 546
	*	Other memory boxes for other variables
	*	

This, in a conceptual sense, is how your Model 100 thinks about variables. The above example shows how three variables with the names HEIGHT, A, and AMT are assigned the values 16, 5, and -546, respectively.

What if you don't assign a value to a variable? Go ahead and try it with the following program:

```
10 PRINT HEIGHT
30 END
RUN
0
OK
```

BASIC automatically assigned the value 0 to the variable. Generally, if you don't assign a value to a variable, BASIC will do it for you — and it always chooses zero.

Rules for Naming Variables

Before we delve further into the topic of variables, we need to discuss the names you can give them. We already used such variable names as HEIGHT, A, and AMT, but not every name is acceptable to your Model 100. For example, try entering and running the following two program lines:

```
10 LENGTH = 26
20 PRINT LENGTH
RUN
?SN Error in 10
OK
```

The variable name LENGTH seems perfectly reasonable (it's similar to the variable HEIGHT that we used earlier), yet your computer responds with SN (syntax) error. There are a number of very specific rules that dictate how variable names can be constructed. You should become familiar with these rules early on in order to save yourself the frustration of frequent error messages.

The first rule is less a restriction than an aid to programming: all letters in variable names can be entered as lowercase *or* uppercase letters. BASIC makes no distinction between the two. For example, "CAT" is the same variable as "cat". However, no matter how a variable is entered, it will always be LISTed in uppercase letters. You can see this for yourself if you write the following program line:

```
10 vetbill = 355.99
```

and use LIST to list it:

```
LIST
10 VETBILL = 355.99
OK
```

LIST always writes out variable names in uppercase letters, no matter how you enter them. (LIST does the same thing with Reserved Words, by the way, as we saw in Chapter 2.) For the sake of consistency and clarity, however, we'll always display our example programs as they would be listed — that is, with variable names written in capital letters.

The second rule is that variable names must begin with a letter, which means that variable names such as 2YEAR or %INTEREST are taboo.

The third rule is that the rest of the variable name (beginning with the second character) can be made up of either letters or numbers but cannot have punctuation or special characters, such as the period (.) or asterisk (*). Examples of legal variable names are PRICE, INTEREST, and R2D2, while examples of illegal variable names are FISH&CHIPS and YEAR.84. (There are a number of special characters that can and often must be used at the *end* of variable names to specify the variable *type*, but we needn't be concerned with these until later in the book.)

The fourth rule is that BASIC on your Model 100 uses only the first two characters in a variable name to identify the variable. Therefore, as far as BASIC is concerned, the variables PRICE and PR are identical. Consider the following program:

```
10 PRICE = 124
20 PRINT PRICE, PR
RUN
 124          124
OK
```

Line 20 PRINTs out the same value (124) for *both* variables, PRICE and PR, though only the variable PRICE was assigned the value 124 in line 10. This can happen only if your Model 100 thinks that PRICE and PR are the same variable.

Because BASIC recognizes only the first two characters, why use variable names *longer* than two characters? There is one important advantage: longer variable names tend to be more meaningful to us than one- or two-letter names. Longer and more self-explanatory variable names make it easier for programmers to recognize and remember what different variables are meant to represent. For example, if you see the variable PR in an unfamiliar program, you may not know what kind of quantity this variable

represents; however, if you see the variable PRICE, you have a pretty good idea that this variable represents the dollar amount of something. BASIC lets us use long variable names as a mnemonic aid, even though the computer recognizes only the first two letters.

The fifth rule is simple yet often troublesome. Earlier in the book we mentioned a list of Reserved Words (see Appendix A). *Reserved Words* are any words used as BASIC commands or instructions, such as BEEP or PRINT. Our fifth rule is this: *you cannot use a Reserved Word as any part of a variable name.* This explains why the variable LENGTH earlier in this section caused a syntax error: look at the list of Reserved Words in Appendix A — you may as well get familiar with it now — and you will find the word LEN. Well, LEN is contained in the first part of the variable LENGTH, so LENGTH is an illegal variable name. Another example is the name COST. If you try to use COST in a program, once again you'll get a syntax error. What's wrong? The word COS appears in the list of Reserved Words.

Examples of Legal and Illegal Variable Names

Here are some examples of "legal" variable names:

```
AB
A4
X
ADDRESS
ZAR023
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Note that two of these variables (the first and last) are identical as far as BASIC is concerned. Sometimes it is helpful to see what you shouldn't do, so we've included Table 6-1 which lists a number of "illegal" variable names with an explanation of why they're illegal and how you might make them acceptable. Notice in the last three examples how easily and surreptitiously Reserved Words can sneak into otherwise healthy variable names.

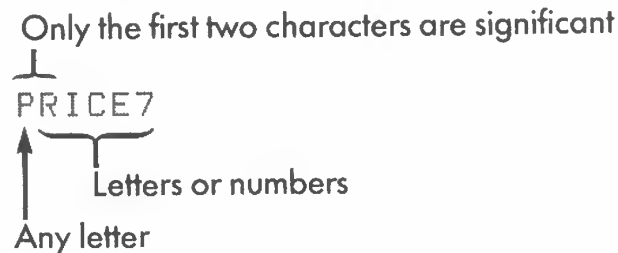
Variable Name	Reason for Illegality	Corrected Version
5DOLLARS	Number as first character	DOLLARS5
PAY.MONTH	Period within the name	PAYMONTH
I&	Special character within name	I
INTEREST	Contains Reserved Word INT	ITEREST
HERFORTUNE	Contains Reserved Word FOR	HERWEALTH
CONTENT	Contains Reserved Word ON	CNTENT

Table 6-1. Examples of illegal variable names

Rules for Numeric Variable Names

1. Variable names are always written out in uppercase by the LIST command, although they may be entered in either lower-case or uppercase.
2. Numeric variable names must begin with a letter.
3. Subsequent characters can be letters or numbers but not punctuation or other characters.
4. Only the first two characters of a variable name are used by Model 100 BASIC to identify a variable.
5. Reserved Words (Appendix A) cannot be part of a variable name.

These rules are summarized in the following diagram:



Arithmetic with Variables Using PRINT

So far we've talked about what a variable is, how to introduce it into a program, and how to assign a value to it. Now we'd like to show you how variables can be used in a simple but rather powerful way.

Consider the following domestic problem. The walls in your living room haven't been painted for ten years, and because you hate painting, you've decided to cover them with wallpaper. Wallpaper is pretty expensive stuff, so you want to buy just the right amount. You decide to write a program on your brand-new Model 100 that determines the number of square feet of wall surface to be covered by wallpaper.

Let's consider just one wall first. (We'll finish the calculations for the other walls at the very end of this chapter.) You measure the wall and find its length to be sixteen feet and its height to be eight feet. Knowing that

area is the product of length and height, you might be tempted to write a program like this one:

```
10 PRINT "Area of wall ="
20 PRINT 16 * 8
30 END
RUN
Area of wall =
  128
OK
```

Well, that does the job. But this program has one serious limitation: it works only for *one* of your walls. In other words, if you wanted to use this program to find the area of an adjacent wall, which undoubtedly has different dimensions, you'd have to rewrite the whole PRINT statement. Worse yet, if you were a wallpaper contractor who wanted to generate a table showing the area of walls having a thousand different sizes, you'd really be in trouble!

So, armed with your knowledge of variables, you head off all such problems by writing the following program:

```
10 LENGH = 16
20 HEIGHT = 8
30 PRINT "Area of wall ="
40 PRINT LENGH * HEIGHT
50 END
RUN
Area of wall =
  128
OK
```

This program really works! Lines 10 and 20 assign the values 16 and 8 to the variables LENGH (remember that LENGTH is illegal) and HEIGHT. Line 40 multiplies the value of LENGH by the value of HEIGHT; it does exactly what it would do if we had asked it to PRINT 16 * 8.

Why is this program using variables any better than the previous one, which contained only numbers? First, it's more versatile, more flexible. The PRINT statement in line 40 calculates the product of *any* length and height. This may not now seem like much of an advantage, because you'd have to change lines 10 and 20 anyway in order to find the area of a wall with different dimensions. But in Chapter 7 we'll show you how to write some BASIC statements that could, when you run your program, ask you for the length and height each time your Model 100 must calculate the area of another surface. Thus, if you use variables, you won't have to change your program at all to find the area of any wall.

To summarize, using variables allows you to write BASIC statements and whole programs that work for any particular value. It's like a cookbook recipe — in which you can vary the ingredients — that works no matter whether you cook just one serving for yourself or a hundred servings for all your friends.

The previous example involved the multiplication of two variables, but, not surprisingly, we can also use variables for addition, subtraction, and division. Try the following example to get a feel for this:

```
10 X = 6
20 Y = 3
30 PRINT X + Y
40 PRINT X - Y
50 PRINT X/Y
60 END
RUN
9
3
2
OK
```

Everything worked as expected. This time, for the sake of variety, we used the rather short variable names X and Y. To many people, these are the “traditional” sort of variables used in algebra.

Printing Numeric Variable Expressions

The PRINT statement will evaluate numeric expressions involving variables and will write the value of the expression to the screen. An example is

```
50 PRINT X/Y
```

which will return a value for the quotient X/Y, using the present values of X and Y to do the division.

String Variables

So far in this chapter we have been working with numeric variables; that is, we have assigned numbers to our variables. However, the remarkable fact is that we can assign letters, words, or even whole phrases to a variable as well! Variables whose values consist of words are called *string variables*, and

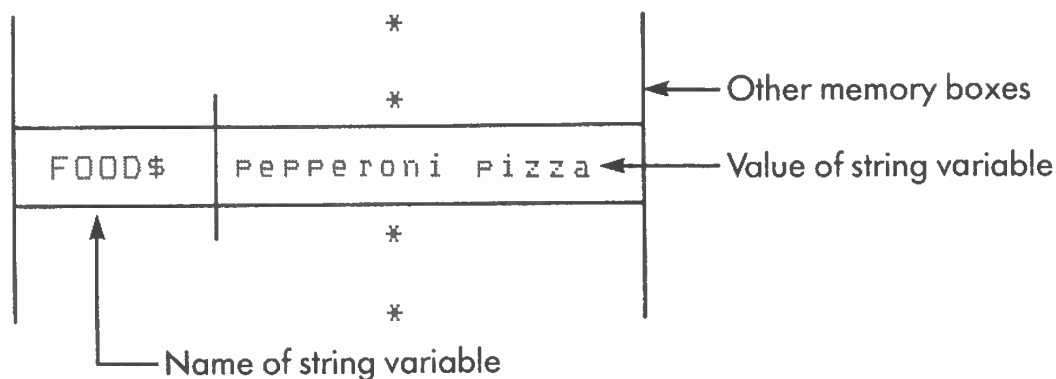
their values are called *string constants*. A string constant is the same thing we called a “string” in previous chapters (see the discussion of PRINT in Chapter 3). Like strings, string constants must be enclosed in quotation marks.

To get started with string variables, let’s write a simple program and run it:

```
10 FOOD$ = "pePPERoni Pizza"
20 PRINT FOOD$
30 END
RUN
pePPERoni Pizza
OK
```

Let’s find out what happened here. Line 10 tells BASIC to put the phrase “pepperoni pizza” in a variable box having the name FOOD\$. In other words, line 10 *assigns* the string constant “pepperoni pizza” to the variable called FOOD\$. The PRINT statement in line 20 then prints the value of the string variable.

You can see that both numeric and string variables are assigned values in pretty much the same way. We can use our earlier box (or house) analogy to visualize how the computer thinks about string variables:



There is, however, one crucial difference between numeric and string variables that has to do with the dollar sign (\$) at the end of the FOOD\$. No, we didn’t put it there because food costs money! Rather, it is a rule that *string variable names must end with a dollar sign (\$)*. The dollar sign signals your Model 100 that it is dealing with a string variable. Your computer must know which type of variable it is dealing with because it reserves memory space for the values of variables, and this memory space is different for numeric and string variables.

If you do try to assign a string constant to a variable without a dollar sign at the end of its name, you'll get an error statement, as the following example demonstrates:

```
10 GREETING = "hello!"
20 PRINT GREETING
RUN
?TM Error in 10
OK
```

The letters *TM* stand for “Type Mismatch” — your Model 100 is telling you that you're mismatching the *type of variable* and the *type of value* in line 10. You'd get the same error message if you tried to assign a numeric value to a string variable.

Assigning String Variables

String constants can be assigned to string variables by means of the equal sign, as shown in the following example:

```
40 BIRD$ = "pelican"
```



Dollar sign at end of variable name indicates that this is a *string* variable.

Only the first two characters of a variable name are used by BASIC to identify the name.

A string constant is a sequence of characters enclosed by quotation marks and can range from 0 to 255 characters long.

A string variable name must end with a dollar sign (\$). Otherwise, the rules for valid string variable names are identical to those for numeric variable names.

An Example Using String Variables

String variables are handy when you want your program to print a message or part of a message that needs to be changed from time to time, such as a list of names, phone numbers, or favorite foods.

Let's try this example:

```
10 FOOD$ = "PePPERoni Pizza"
20 PRINT "My favorite food is "; FOOD$
30 FOOD$ = "tripe"
40 PRINT "My favorite food is "; FOOD$
50 END
RUN
My favorite food is PEPPERoni Pizza
My favorite food is tripe
OK
```

Well, you can take your choice! The point is that string variables enable us to make the same PRINT statement (lines 20 and 40) print different results. Both PRINT statements contain the string constant "My favorite food is " and the string variable FOOD\$, but what PRINT returns (that is, what it does when the program runs) depends on the value of the variable. In a more sophisticated program of the kind you'll learn to write in Chapter 7, the value of FOOD\$ might be determined by the program itself or by some input from the user so that one PRINT statement would be able to produce many different outputs.

Notice the use of the semicolon in the PRINT statement: it causes the variable's value (the string constant) to be printed right after the string "My favorite food is ". In fact, had we left out the space after the word *is* in "My favorite food is ", the two strings would have been run together to read "My favorite food istripe".

Concatenation — Adding Strings

As we have seen, we can perform arithmetic operations on numeric variables. Can we perform any similar operations on string variables? It would hardly make sense to divide two strings, but, amazingly, BASIC does allow us to *add* two string variables or two string constants. Seeing is believing, so let's try this program:

```
10 F1$ = "PePPERoni "
20 F2$ = "and mushroom Pizza"
30 PRINT F1$ + F2$
40 END
RUN
PePPERoni and mushroom Pizza
OK
```

What has happened here is that the values of the variables F1\$ and F2\$ have been printed right next to each other, or joined together. The plus sign

(+) between the string variables stands for “concatenate”, which simply means to join or to link. So line 30 prints the linked values of the variables F1\$ and F2\$.

Note our use of numbers in variable names to differentiate between two variables. If you are tempted to use the longer variable names FOOD1\$ and FOOD2\$ in place of F1\$ and F2\$, remember that because BASIC reads only the first two letters of a variable name, it would consider FOOD1\$ and FOOD2\$ to be identical. (Though concatenation may not yet seem a very useful process, we’ll use it to good advantage in a later chapter on the manipulation of strings.)

Concatenation

Concatenating two string variables (or string constants) by means of a plus sign (+) joins their values without adding any extra spaces. For example, the statement

```
45 PRINT "hot" + "dog"
```

returns the value hotdog

Numeric Expressions

Recall the program in which we used the following PRINT statement to calculate the area of a wall:

```
40 PRINT LENG * HEIGHT
```

Using PRINT in this way to evaluate expressions is perfectly acceptable, but it is not always the best way, nor is it always possible. More complex problems usually involve a whole sequence of expressions, most of which you may not actually want to print. It is more convenient (and sometimes necessary) to define a new variable in terms of old variables.

Let's rewrite our wall area program in the following way:

```
10 LENGH = 16
20 HEIGHT = 8
30 AREA = LENGH * HEIGHT
40 PRINT "area of wall ="
50 PRINT AREA
60 END
RUN
area of wall =
  128
OK
```

The significant new BASIC statement is line 30: it defines a new variable, AREA, which equals the product of the variables LENGH and HEIGHT. This makes perfect sense, because that's the meaning of *area*. Your computer interprets line 30 to mean "find the product of the value of the variables LENGH and HEIGHT and assign it to a variable called AREA". In terms of our analogy of computer memory boxes, we might imagine that a new memory box called AREA is added to the previously defined boxes LENGH and HEIGHT. The central computer "brain", called the Central Processing Unit (CPU), calculates the product of 16 and 8 and puts the result into the value part of the variable box AREA. The PRINT statement in line 50 now simply prints the value of the variable AREA, which was already calculated in line 30.

Working with Several Numeric Expressions

To get a better feel for the power of using variables, let's also ask our computer to find the *cost* of the required wallpaper. We'll introduce a new variable called PRICE, which represents the price of the wallpaper in dollars per square foot. Assuming that the price is \$6 per square foot, we can write our program this way:

```
10 LENGH = 16
20 HEIGHT = 8
30 PRICE = 6
40 AREA = LENGH * HEIGHT
50 CST = PRICE * AREA
60 PRINT "Area=", "Cost($)=
70 PRINT AREA, CST
80 END
OK
RUN
Area=          Cost($)=
  128          768
OK
```

This is pretty neat stuff! The first new line is line 30: it assigns the value 6 (dollars) to the variable PRICE. Line 50 multiplies the value of PRICE by the value of AREA and assigns this product to the variable CST. (CST represents “CoST” — remember that COST is an illegal variable name because it contains the Reserved Word COS.) PRINT in line 70 simply writes out the values of AREA and CST (with the fourteen-character-wide zone spacing produced by the comma — see Chapter 3).

The main point here is to demonstrate how a problem — and the corresponding program — can be broken up into small units. Each unit represents a phase in the process in which variables are defined in terms of one another. The first such unit finds the area by means of line 40; the second, in line 50, finds the cost in terms of price and area. This problem of finding the cost of wallpaper is simple enough that we certainly could have condensed lines 40, 50, and 70 into one BASIC statement. For example, we might have written

```
40 PRINT LNTH * HEIGHT * PRICE
```

to take care of the whole job. However, our original program is more “transparent” in that we can more easily see what’s going on, and we get the fringe benefit of finding (and printing) the area of the wall as well as the cost of wallpapering it.

Assigning New Variables in Terms of Old Variables

We can define new variables in terms of the operation between other variables by means of the equal sign (=) as in the following example:

```
AREA = LNTH * HEIGHT
```

To the computer this means:

“To the variable AREA, assign the value obtained by multiplying the value of LENGTH by the value of HEIGHT.”

All the arithmetic operations — addition, subtraction, multiplication, and division — may be involved in these operations on variables.

Rules of Order — Expressions with Three or More Variables

The *order* in which you write the variables in an arithmetic expression can influence the results. As an analogy, suppose that some new friends invite you over for dinner at their place. You haven't been to their house before, so you are given the following instructions: "Turn left at the fifth stop sign, go another two blocks and turn right, and then find the first pink house on your left". These directions have two important aspects. The first has to do with the individual "operations" you need to perform, such as "go another two blocks and turn right". The second has to do with the *order* in which you carry out the instructions. If you get either aspect of your directions mixed up, you'll never get to dinner. The point is that in addition to getting the individual operations right, you must also do the operations in the correct order. Of course, we're really talking not about finding your way through town but about rules for evaluating arithmetic expressions involving more than two operations.

To be more specific, let's look at the following program:

```
10 A = 2
20 B = 3
30 C = 4
40 D = A + B * C
50 PRINT D
60 END
RUN
  14
OK
```

Line 40 defines variable D in terms of two arithmetic operations on variables A, B, and C. Which operation did your computer do first? Did it first add A to B and then multiply that sum by C? Or did it first do the multiplication and then the addition? We can find out what your computer did by checking the arithmetic ourselves. If we add first and then multiply, we get 20, but if we multiply first and then add, we get 14 — the same answer the computer got.

Computers always do all multiplications before any additions, even though, as in this example, the addition appears first in our line of instructions. What about subtraction and division? Subtraction is similar to addition and division is similar to multiplication, so the rule is that *all multiplications and divisions in an arithmetic expression are evaluated before additions and subtractions*. We should note here, however, that we can override this rule by using parentheses. We'll explain this in the last section of this chapter.

Rules for Evaluating Arithmetic Expressions:

All $\left\{ \begin{array}{c} \text{multiplications} \\ \text{and} \\ \text{divisions} \end{array} \right\}$ are carried out before $\left\{ \begin{array}{c} \text{additions} \\ \text{and} \\ \text{subtractions} \end{array} \right\}$

provided that no parentheses are used.

Let's add a few more lines to our previous program to include subtraction and division. The result looks like this:

```
10 A = 2
20 B = 3
40 C = 4
50 D = A + B*C      ← 2 + 3*4 = 2 + 12 = 14
60 E = A/B + C      ← 2/3 + 4 = .66666666666667 + 4 = 4.6666666666667
70 F = A*B - A/C    ← 2*3 - 2/4 = 6 - .5 = 5.5
80 PRINT D,E,F
90 END
RUN
14      4.66666666666667
5.5
OK
```

We suggest that you evaluate lines 50, 60, and 70 yourself to see if the computer did it right.

Overriding the Rules with Parentheses

Using parentheses in arithmetic expressions allows us to override the rules described in the previous section. Consider the following program:

```
10 A = 2
20 B = 3
30 C = 6
40 D = A + B*C
50 E = (A + B)*C
60 PRINT D,E
70 END
RUN
14      20
OK
```

Lines 40 and 50 are identical except for the parentheses. The parentheses in line 50 tell the computer to “add A and B first and then multiply that sum by C”. In general, parentheses instruct your computer to *take care of the operations inside the parentheses first* and then proceed, using the rules of order discussed in the previous section. Parentheses are a very useful tool: they give you great flexibility in writing expressions. Also, if you forget the rules of order, you can always use parentheses to be sure to get what you want.

Order of Operations with Parentheses

Parentheses in arithmetic expressions are evaluated in the following way. First, expressions inside the innermost parentheses are evaluated; these expressions are then treated as values. Then the next level of parenthetical expressions are evaluated; these in turn are treated as values for the next stage in this process. This procedure is continued until a single value results.

Summary

In this chapter we have focused on a very important programming tool known as the *variable*. We first explained the concept underlying variables — that variables allow us to plug different values or pieces of information into the same program line, thus making the program more general, more flexible, and more understandable. The value (or constant) assigned to a variable may also be reassigned within the program or, as we will see later, by input from a user interacting with the program. We also explained the rules that dictate how variable names must be constructed. One such rule is that string variable names must always end in a dollar sign; this dollar sign differentiates a string variable from a numeric variable. The values assigned to string variables, called *string constants*, must always be enclosed in quotation marks.

Though we did not present any new BASIC instructions in this chapter, we did explain several kinds of operations we can perform using variables. For example, we can concatenate (or link) string variables with the plus sign. We also saw how we can add, subtract, multiply, and divide numeric variables. In the process, we saw that the order in which we perform BASIC arithmetic operations is the same as in algebra (multiplication and division

come first, followed by addition and subtraction) but that we can override this natural order by using parentheses. Finally, we saw how we can define new variables in terms of old ones. Defining variables in terms of one another in this way allows us to break a problem down into manageable parts.

Exercises

1. Suppose you must cover the walls of a room with wallpaper. Write a program that determines the total number of square feet of wallpaper required and the total cost of the wallpaper. Here is the relevant information: the room is 16 feet long, 12 feet wide, and 8 feet high, and the price of the wallpaper is \$6 per square foot. The program should clear the screen and create a well-formatted output.

2. In Chapter 3 you learned how to place a character anywhere on your Model 100 screen by using the PRINT @ statement. The number following the "@" sign is what we called the *character coordinate*. Write a BASIC statement that returns the character coordinate, given the row and column position of the character to be placed on the screen. Row 1 and column 1 define the upper left corner of your screen.

Solutions

1. Here's one possible solution to the wallpaper problem:

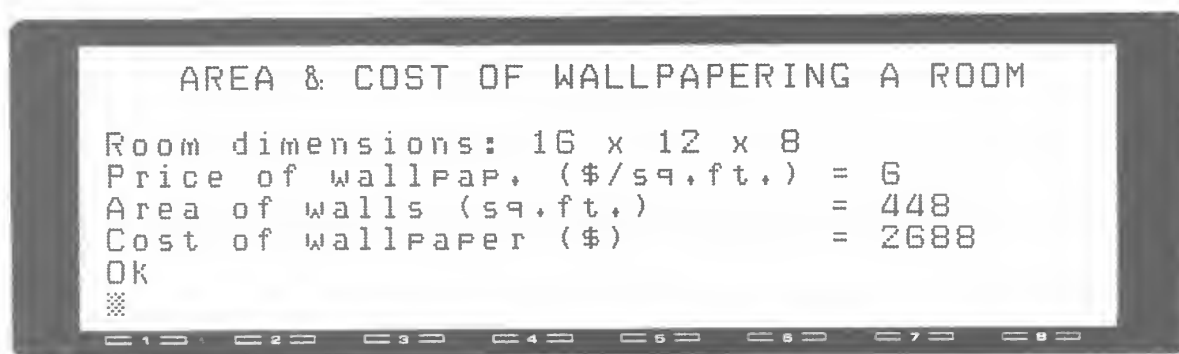
```
100 REM--*****
110 REM--      NAME: "WALLPA"
120 REM--Prgm finds AREA of walls and
130 REM--CST (cost) of wallpap. a room
140 REM--*****
150 CLS
160 REM--VARIABLE ASSIGNMENT-----
170 REM
180  L = 16
190  W = 12
200  H = 8
210  PRICE = 6
220 REM
230 REM--MAIN PROGRAM-----
240 REM
250  A1 = L * H
```

```

260 A2 = W * H
270 AREA = (A1 + A2) * 2
280 CST = PRICE * AREA
290 REM
300 REM--OUTPUT OF PROGRAM-----
310 REM
320 PRINT "    AREA & COST OF WALLPAPERING A ROOM"
330 PRINT
340 PRINT "Room dimensions:";L;"x";W;"x";H
350 PRINT "Price of wallpap, ($/sq.ft.) =" ;PRICE
360 PRINT "Area of walls (sq.ft.)          =" ;AREA
370 PRINT "Cost of wallpaper ($)           =" ;COST
380 END

```

The program's output on your screen looks like this:



We suggest that you take some time to look over this program to see if everything makes sense to you. There are a few remarks we'd like to make about it.

Notice the use of REM statements and the indentation of the BASIC instructions right after the line numbers. Both of these help make the program easier to read and understand.

A word about what we called the MAIN PROGRAM. The variable A1 is the area of one of the walls. A2 is the area of the adjacent wall, which is a different size. The total area of all the walls, which we called AREA, is given by twice the sum of A1 and A2 — that accounts for four walls.

Also notice that our program appears as it would be printed on a printer using LLIST — that is, some of our listed lines are longer than forty characters. If you LIST this program on your screen, some of the longer lines (for example, lines 340-370) will be folded over.

2. The following statement calculates a value for the character coordinate and assigns this value to the variable CC:

```

30 CC = (COL - 1) + (ROW - 1)*40

```

The variables COL and ROW represent the COLUMN and ROW position, respectively. This line is very handy in a program that places a character on your screen at a given row and column position. For example, the program

```
10 ROW = 20
20 COL = 4
30 CC = (COL - 1) + (ROW - 1)*40
40 PRINT @ CC, "X"
50 END
```

places an “X” in the center of your screen. We’ll put this technique to good use in a later chapter on character graphics.

7

INPUT to Your Program

Concepts

- INPUT to numeric and string variables
- INPUT with prompt
- INPUT to multiple variables
- Error message from INPUT
- Interrupting INPUT

Instructions

- INPUT, LINE INPUT

Computer programs can generally be thought of as being divided into three stages. The first stage is *input*, in which the program gathers the information it needs from the outside world. The second stage is the *main program*, which does all the manipulating and calculating needed to perform the main task for which the program was originally written. The last stage is *output*, which displays the results obtained by the main program in understandable form.



In previous chapters you have learned much about the last two stages: variables and the various operations on them provide us with many of the tools required in a typical main program, while PRINT and PRINT-related statements give us very powerful tools for displaying output.

In this chapter, we introduce you to the first stage — input. In case you're wondering why we're covering the first stage last . . . well, we do have a reason. We needed to wait until you learned something about variables and how to use PRINT so that you could see what information we actually put into a program.

What exactly do we mean by *input* to a program? Recall the wallpaper program at end of Chapter 6. We could provide our program with the dimensions of the room and the price of the wallpaper only by direct assignment statements like

```
10 LENGTH = 16
```

This is fine as far as it goes, but there's a problem: if you want to run that program again for a *different* room size or price of wallpaper, you must rewrite those particular assignment statements, making the changes inside your program. Wouldn't it be a lot better and more convenient if your program would *ask* you, the user, a question like: "What values for room size and wallpaper price should I use this time?" Then you could RUN the program over and over again with different input information — without modifying the program every time.

The Basic INPUT Statement

The INPUT statement is exactly what we need to solve this problem. INPUT is BASIC's way of asking the user for information.

Inputting Numeric Variables

To see how INPUT can be used with numeric variables, let's dive right in and enter the following program:

```
10 PRINT "enter price ($)"
20 INPUT PRICE
30 PRINT PRICE
40 END
```

When you RUN this program, the following appears on your screen:

```
RUN
enter price ($)
? █                ← Blinking cursor
```

The blinking cursor right after the question mark means "I'm ready for you now; enter a value". So go ahead, enter a value — say, 15.25.

```
enter price ($)
? 15.25
  15.25
OK
```


What's new in this program is the INPUT statement in line 20. The INPUT statement is responsible for the question mark (?) and the blinking cursor that request input from you. Your computer will sit there and wait until you enter something (or until you do something drastic, like turn the power off!). We entered the value 15.25, which is immediately assigned to the variable PRICE — the variable listed right after INPUT in line 20. When it is PRINT's turn to do its job, it displays the value of PRICE, namely, 15.25. Pretty straightforward!

RUN the program again, but this time enter a different value in response to the question mark:

```
RUN
enter Price ($)
? 23500.99
  23500.99
OK
```

As you can see, whatever you enter in response to the question mark resulting from INPUT really does get assigned to the variable PRICE.

So here we have what we were looking for — a simple method of passing input information from you, the user, to the program, *without* having to cast this information in concrete by using assignment statements within the program. Your computer talks to you with PRINT, whereas it *listens* to you with INPUT. You can see why *INPUT* is one of the most important words in the BASIC vocabulary.

The Simple Form of INPUT for Numeric Variables

The statement

```
30 INPUT PRICE
```

results in the following output:

```
? █
```

← Blinking cursor

The cursor waits for the user to enter a numeric value. The entered value is assigned to the variable PRICE.

Inputting String Variables

By now you won't be too surprised to learn that we can also INPUT string variables. Try this program:

```
10 PRINT "enter your name"
20 INPUT NAM$
30 PRINT NAM$
40 END
```

Running this program causes the already familiar question mark and blinking cursor to appear on your screen:

```
RUN
enter your name
? █ ← Blinking cursor
```

As before, the question mark and cursor are your computer's request for input. Suppose we respond with Dietrich Buxtehude (he can't do it himself — he died in 1707!). The complete output of the program looks like this:

```
RUN
enter your name
? Dietrich Buxtehude
Dietrich Buxtehude
OK
```

As you can see, inputting values to string variables is very similar to inputting values to numeric variables. The only difference is in the variable type. The dollar sign (\$) at the end of the variable NAM\$, right after INPUT, tells BASIC to expect a *string* constant rather than a numeric constant. That's in line with what we already know about assigning variables of different types. What may be somewhat surprising to you is that even though the phrase "Dietrich Buxtehude" is a string constant, we didn't put any quotation marks around it when we input it. You *can* bracket a string entry with quotation marks, but in general this is not required. (In the next section we'll show you a case in which such quotation marks do serve a necessary function.)

You may also have wondered why we used NAM\$ as our variable name and not the more obvious NAME\$. Go ahead and try it — if you want to get an "SN Error in 20" response! (We have to admit we fell for this too.) The key to the mystery is the list of Reserved Words in the Appendix A: NAME is one of them! We can't use variable names that are also Reserved Words.

Interrupting INPUT with **SHIFT** **BREAK**

There may be times when you don't want to respond to the INPUT question mark with the type of response we've just discussed. For example, if you are dealing with a lengthy program and realize that you just entered something you didn't want, you may wish to break out of this particular RUN of the program and start over again. Fortunately, there is a way to do this: press the **SHIFT** **BREAK** key combination. Let's try it once with the following familiar program. Enter RUN to get the usual question mark:

```
10 INPUT NUMBER
20 PRINT NUMBER
30 END
RUN
?
```

Now press **SHIFT** **BREAK** and your program output will look like this:

```
RUN
? ^C
Break in 10
OK
```

This tells you two things: first, the “break” or interruption occurred in line 10, which may be handy to know in a complex program; second, BASIC is once again at your service, as indicated by the Ok.

Recall from Chapter 4 that you can use the **SHIFT** **BREAK** combination to interrupt your program at *any* stage of its execution. This is a potent key combination — especially when you need to get out of trouble!

INPUT with Instructions

So far we've used a PRINT statement right in front of the INPUT statement to tell the user what information INPUT wants. For example, if you had wanted to INPUT a value for your age, you might have written the following:

```
10 PRINT "how old are you?"
20 INPUT AGE
30 PRINT AGE
40 END
```

Here PRINT's function is to let you know that when INPUT's question mark appears, you should enter your age and not some other possibly interesting but irrelevant information.

This approach works just fine, of course. But there's a shortcut: namely, the `INPUT` statement with what is called the "prompt". The prompt is a string constant within the `INPUT` statement that replaces the initial `PRINT` statement immediately before `INPUT`. The following program does exactly the same thing as the previous one, but it does so without the initial `PRINT`:

```
10 INPUT "how old are you?"; AGE
20 PRINT AGE
30 END
RUN
how old are you? ⌘
```

As before, the question mark and the cursor invite your response. The prompt is the string "how old are you?" It always follows right after the word *INPUT*. When `INPUT` is executed, it first prints out the prompt, then displays the question mark and cursor. To finish this example, let's enter a number:

```
RUN
how old are you? 41
  41
OK
```

There. `INPUT` with a prompt behaves exactly the same way as `INPUT` without a prompt.

INPUT with Prompt

The following statement illustrates the syntax of an `INPUT` statement with a prompt:

```
10 INPUT "how old are you?"; AGE
```

└──────────────────┘ └─┘
The prompt The variable

When this line is executed in a program, it requests user input in the following form:

```
how old are you? ⌘      ← Blinking cursor
```

INPUT with Multiple Variables

So far we've entered just one value with each INPUT statement. Some programs, however, require the INPUT of a lot of data, such as lists of stock prices, test scores, and the like. These programs would require as many INPUT statements as the number of values you need to enter. In cases like this, wouldn't it be helpful to be able to enter a lot of data with just *one* INPUT statement?

Indeed, the versatile INPUT allows you to do just that. You need only list the variables (to which the user is to assign values), separated by commas, right after the word *INPUT*. Let's try it out:

```
10 INPUT "A word, and a number"; WRD$, NUMBER
20 PRINT WRD$, NUMBER
30 END
RUN
A word, and a number? cello, 6500
cello          6500
OK
```

Our entry in response to the INPUT question mark was the word *cello*, then a comma, and the number 6500. The first entry, "cello", is assigned to the first listed variable WRD\$, and the second entry, 6500, is assigned to the next listed variable, NUMBER. In other words, the order in which values are entered in response to INPUT should be identical to the order in which variables are listed after INPUT. That makes sense.

In the above example, we typed in one value, then a comma, then the second value, and finally we pressed **ENTER**. But there's another way to enter several values in response to one INPUT question mark: instead of entering all values at the same time, we can enter each value separately. Try it with this program:

```
10 INPUT "Enter 4 numbers"; N1,N2,N3,N4
20 PRINT N1,N2,N3,N4
30 END
RUN
Enter 4 numbers? 5      ← We entered 5
?? 3                   ← Then 3
?? 7                   ← Then 7
?? 2                   ← Then 2
  5                     3
  7                     2
OK
```

As you can see, after you've entered the first number (5 in our example), your Model 100 responds with *two* question marks. That's the signal to you to keep on entering values. After you've entered the fourth value, line 20 will print out all the values you entered.

How many values can you enter using INPUT with either of the methods described above? The maximum length of the logical BASIC line, 255 characters, is the only limitation. So if you use short variable names, you'll have room for quite a few variables!

INPUT's Error Message — What Can Go Wrong, Wrong, Wrong

In our previous examples, our responses to the INPUT question mark have been perfectly acceptable to BASIC since we haven't gotten any error messages. However, there are two ways we *can* make an inappropriate response.

One way to make an entry error is to enter the wrong *type* of variable. Let's run the previous program once again, but with the entry shown:

```
RUN
Enter 4 numbers? three
?Redo from start
? 5,999,1,3
  5          999
  1          3
Ok
```

What went wrong with our first entry? The phrase “?Redo from start” means “Start over again, and do it right this time!” It's not hard to see where we went wrong: we entered a *string* when BASIC expected a *number*. Similarly, if we enter a number when BASIC is expecting a string, we'll get the same message. BASIC always responds with “?Redo from start” — giving us a chance to do it over — whenever there is a *type mismatch* between the value entered and the variable to which we try to assign it. (Computers can be very petty about things like that!)

Another error message results when the number of values we're trying to input exceeds the number of variables listed after INPUT. For example, run the previous program once again, this time with five entries:

```
      RUN
Enter 4 numbers? 1,2,3,4,5
?Extra ignored
  1          2
  3          4
Ok
```

We entered five values, whereas the INPUT statement only has four variables listed. What happens is fairly self explanatory: BASIC says, in effect, "You entered too many values, so I'll just ignore the extra values"; then it proceeds, using only as many values as can be matched with the variables listed after INPUT. BASIC on your Model 100 is really very forgiving in this respect.

INPUT of Strings with Commas

The examples in the previous section show that multiple values entered in response to the INPUT question mark and BASIC prompt should be separated by commas. In general, BASIC uses the comma to separate different values as well as different variables. This raises the question: "How, then, do we input strings that happen to have commas in them?"

To illustrate the problem, let's rerun the following familiar program:

```
10 INPUT "Enter your name";NAM$
20 PRINT NAM$
30 END
RUN
Enter your name? Buxtehude, Dietrich
?Extra ignored
Buxtehude
OK
```

Apparently, only the first part of our entry, Buxtehude, was assigned to the variable NAM\$. The "Extra" that was ignored was the first name of Mr. Buxtehude, which is Dietrich. The culprit, of course, is the comma, which separates the two parts of the name. As we saw in the previous section, the comma tells BASIC that *two* strings are being entered, whereas only one variable is listed after INPUT — NAM\$.

In situations like the above example, we really do want to enter both parts of the name, including the comma that separates them. There are several ways to do this. One method is to type quotation marks around the string you are entering. Try it with our previous program:

```
RUN
Enter your name? "Buxtehude, Dietrich"
Buxtehude, Dietrich
OK
```

Great, that did the trick. Notice that although you entered quotation marks, they didn't get printed. PRINT never shows you the quotation marks that define the boundaries of a string, but whatever is "inside" the quotation marks gets treated literally by both INPUT and PRINT.

LINE INPUT — For Strings Only

Using quotation marks to enter a string that includes commas is a simple solution but not necessarily the best one: it puts the whole burden on the user. It's inconvenient to have to use quotation marks, and they're easy to forget (if this author's experience is any indication). BASIC gives us another option — the LINE INPUT statement.

LINE INPUT does almost the same thing as INPUT, but its function is more specialized: it can *only* be used to input a *string* constant. It avoids the comma problem of INPUT by allowing only *one* entry for each LINE INPUT statement. Therefore, *all* characters, including the comma, can now be interpreted as characters that are part of the string constant. Also, LINE INPUT doesn't automatically present the user with a question mark. If you want one, the programmer has to write it explicitly into the prompt.

Let's try the following example, which illustrates the use of LINE INPUT:

```
10 LINE INPUT "Name and Prof.? "; NP$
20 LINE INPUT "Birthdate-----? "; BD$
30 PRINT
40 PRINT NP$ ", "
50 PRINT "was born on " B,DATE$
60 END
RUN
Name and Prof.? D. Buxtehude, composer
Birthdate-----? July 8, 1637
```

```
' D. Buxtehude, composer,
was born on July 8, 1637
OK
```

If you've never heard his wonderful organ music, you've missed something (*if* you like organ music, that is)! Notice the commas in our responses to the prompts of LINE INPUT: they're reproduced by PRINT exactly as entered, so we know that they are really part of the string values assigned to the two string variables. Notice also that we included our own question marks within the prompts in lines 10 and 20, because LINE INPUT does not automatically display the question mark.

LINE INPUT

The syntax of LINE INPUT is identical to that of INPUT *except* for the following differences:

1. LINE INPUT accepts *only* string constants. *All* characters entered in response to the prompt, including commas, leading blanks, and quotation marks, are included in the string constant.
2. Only *one* string constant can be entered with each LINE INPUT statement.
3. LINE INPUT does *not* return a question mark as INPUT does.

The variable listed after LINE INPUT must be a string variable (of course!).

Summary

In this chapter you've learned about the *input* stage of a BASIC program. You've learned how to use the INPUT and LINE INPUT statements to enable the user to enter values into a program — *without* having to make changes within the program itself. You can now write programs that can be RUN over and over again with different input values.

You now have at your disposal many of the fundamental tools for programming in BASIC. You know how to use variables, how to INPUT values, how to manipulate variables within the main program, and how to PRINT out the results. But you are still missing one essential tool: the ability to break out of the order of executing program lines, as dictated by your line numbers, that is, the ability to *branch*. That is the principal topic of Chapter 8.

Exercises

1. Write a program that asks the user for a temperature in degrees Fahrenheit and then calculates and prints out the temperature in degrees

Centigrade. To make the conversion to degrees Centigrade, subtract 32 from the temperature in degrees Fahrenheit and then multiply the result by 5/9.

2. As a way to practice using a single INPUT statement to enter several values, write a program that asks the user for five daily prices of a stock and then calculates their average. Use REM statements to identify the three principal stages of the program: input, main program, and output.

Solutions

1. The following program converts degrees Fahrenheit to degrees Centigrade:

```
10 REM--NAME: "FTOC"-----
20 REM--CONVERS. FROM DEG. F -> DEG. C-
30 REM
40 CLS
50 INPUT "Enter degrees Fahrenheit"; F
60 C = (F - 32)*5/9
70 PRINT "Degrees Centigrade ="; C
80 END
RUN
Enter degrees Fahrenheit? 451
Degrees Centigrade = 232.77777777778
OK
```

2. The following is the solution to the stock-averaging problem:

```
100 REM--NAME: "STOCK"-----
110 REM--STOCK AVERAGING PROGRAM-----
120 REM
130 CLS
140 PRINT "    PRGM AVERAGES ENTERED STOCK VALUES"
150 PRINT
160 REM
170 REM--PROGRAM INPUT-----
180 REM
190 LINE INPUT "Company name? "; CO$
200 INPUT "5 stock prices"; P1,P2,P3,P4,P5
210 REM
220 REM--MAIN PROGRAM-----
230 REM
240 PAVERAGE = (P1+P2+P3+P4+P5)/5
250 REM
```

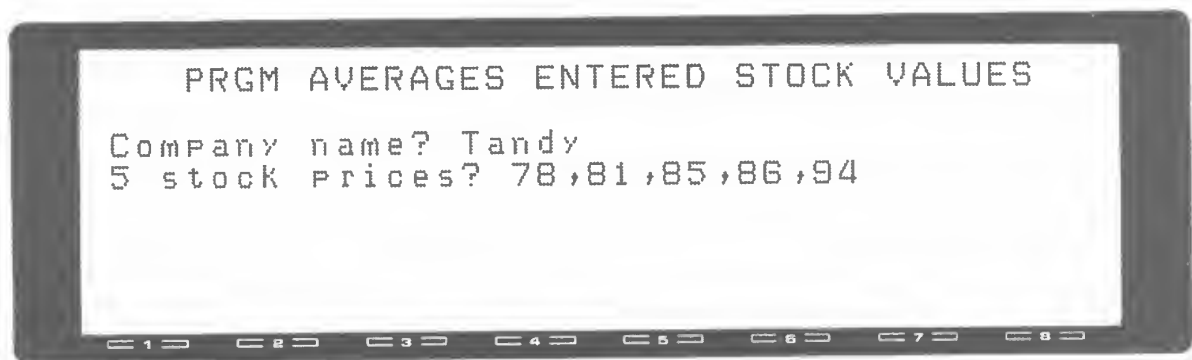
```

260 REM--PROGRAM OUTPUT-----
270 REM
280 CLS
290 PRINT "          STOCK AVERAGING PROGRAM"
300 PRINT
310 PRINT "Company: "; CO$
320 PRINT "5 day stock average ="; PAVERAGE
330 END

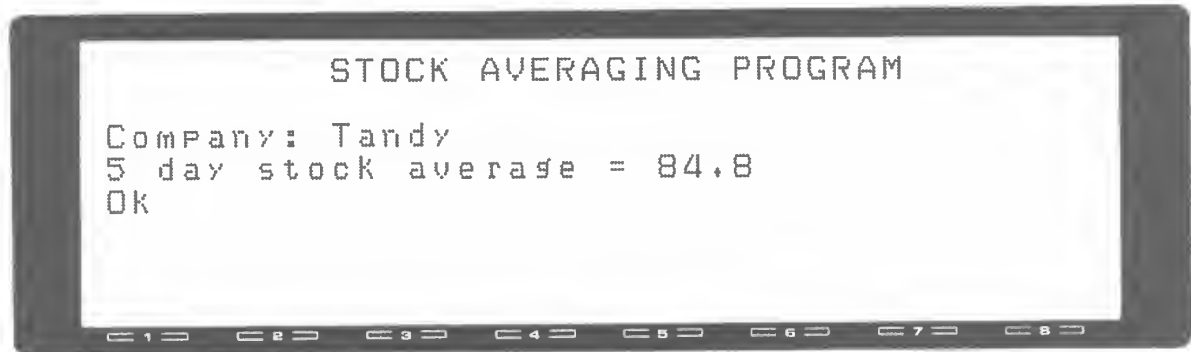
```

Notice that some lines in this program are longer than forty characters. If this program is listed on your screen, these lines will appear folded over. The program as listed here is the way it would be printed out on a printer using the LLIST command.

When you enter RUN, this program clears the screen and asks for user input:



The instant the **ENTER** key is pressed after the last number, the following appears on your screen:



A good week for Tandy! Note that because we used the LINE INPUT statement in line 190, we could enter a company name like "Meryl, Linch, and Finch" — a name with commas in it. Also, note the utility of being able to enter a lot of numbers with one INPUT statement.

8

Branching and Decisions

Concepts

- Unconditional and conditional branching
- Looping
- Relational operators
- Logical operators

Instructions

GOTO, FOR...NEXT, FOR...STEP...NEXT, IF...THEN, IF...THEN...ELSE

All the programs you've written so far are executed by the computer in an order dictated by the line numbers; that is, your computer reads and executes the first line, then the second, and continues this linear process until it reaches the last line. But linear program structures have their limitations. In this chapter we'll discuss two powerful concepts that lie at the heart of programming: looping and branching.

It is often desirable to be able to repeat one portion of a program over and over again, each time with different values. That's just the kind of work we humans find most tedious; the computer, however, does repetitious work with great speed and reliability. This type of repetition, called *looping*, is one of the powerful techniques we'll introduce in this chapter.

It is often also important to be able to *branch* or jump to different parts of a program. This branching is frequently based on some sort of decision. Before you leave your house in the morning, you decide what you'll wear on the basis of the weather. If it is raining, you'll probably wear your boots and other rain paraphernalia, but if the sun is shining, you'll go light. Computer programs frequently need to make similar kinds of decisions.

In this chapter we explore these nonlinear ways to write programs. Looping, branching, and decision making are some of the important and powerful tools a programmer has at his or her disposal.

The GOTO Statement — No Two Ways About It

Of all the BASIC statements related to branching and looping, the GOTO statement is the easiest to understand. To see how it works, enter the following program:

```
10 GOTO 30
20 PRINT "hello, I'm line 20"
30 PRINT "hello, I'm line 30"
40 END
RUN
hello, I'm line 30
OK
```

There are two PRINT statements, but only the second (line 30) was executed! What happened to line 20? The responsible party is, of course, the GOTO statement in line 10. The statement GOTO 30 tells your computer to “go directly to line 30; do not go to line 20 as you normally would”. (Yes, it’s just like in “Monopoly”: “go directly to jail, do not pass go”, and so on.) Figure 8-1 shows how the program is executed. GOTO 30 causes the program to *branch* to line 30, bypassing line 20. This branching is *unconditional* — there simply isn’t any choice.

The GOTO Statement

The statement

```
10 GOTO 30
```

will cause program execution to branch to the specified line number (30 in this example).

Well, that’s really all there is to GOTO. But what good is it? After all, we only caused the program to skip line 20; if that’s all we wanted to do, we needn’t have written line 20 — or line 10, for that matter.

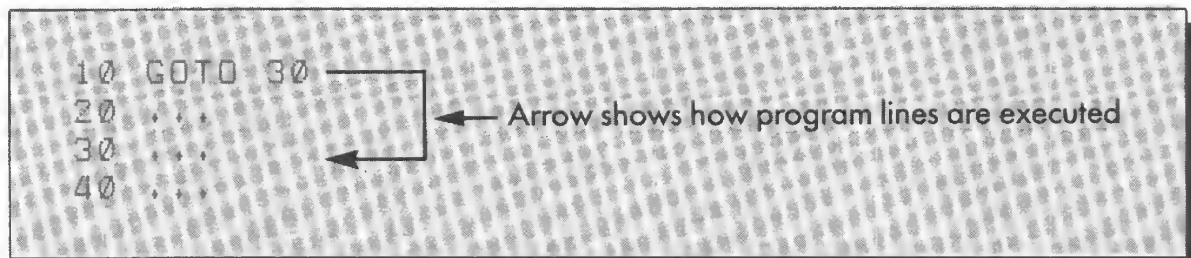


Figure 8-1. Execution of a GOTO statement

As you may have guessed, however, there's more to it. The GOTO statement can be very valuable in conjunction with other programming tools. For example, later in this chapter we'll demonstrate the effective use of GOTO in conjunction with what is called an IF statement.

The Endless Loop — Doing Things Over and Over Again

Let's use GOTO to explore the concept of the *loop*. We used GOTO above to jump forward (to a higher line number) in our program; but we can also use it to jump back to an *earlier* program line. Try the following program:

```
10 PRINT "happy birthday, Ludwig"
20 BEEP
30 GOTO 10
40 END
```

Running this program will cause your Model 100 to beep and to PRINT out

```
RUN
happy birthday, Ludwig
happy birthday, Ludwig
happy birthday, Ludwig
*
*
*
```

over and over again. The “beeps” and the “happy birthday, Ludwig”s keep coming. Is there no end to this? Barring power failure, equipment malfunction, or programmer intervention, the program will run forever! This is called an *endless loop*. Though we have intentionally created this endless loop, such loops can sometimes happen inadvertently, causing real headaches for programmers and users, not to mention the computers themselves (they don't smoke or say “tilt” the way cartoons suggest, but a large system can “crash” — that is, become totally useless to anybody because it's so busy looping).

How does this loop work? First, line 10 prints “happy birthday, Ludwig”. Next, line 20 causes a BEEP for approximately one quarter of a second. The GOTO statement in line 30 then says “go to line 10” — that is, start all over again! So here we go: another “happy birthday, Ludwig”, another BEEP, and GOTO start once again. So 'round and 'round we go — that's our loop. Program execution never gets to the END in line 40. Figure 8-2 helps us visualize this endless loop.

Although this particular loop is rather frivolous, it does illustrate the concept. We can transform such a loop into a useful and truly powerful tool by doing two things: (1) we replace the body of the loop (statements executed in one cycle of the loop) by something more interesting than just a PRINT and a BEEP — that is, really make the loop *do* something; (2) we *control* the loop so that it does only a *specified number of cycles*.

*Interrupting an Endless Loop — The **PAUSE/BREAK** Key*

How do we escape this endless loop? Turning off your Model 100 doesn't help; the instant you turn it back on, your program will continue to run as if you never turned it off.

There are two ways you can interrupt an endless loop or, for that matter, any program. One method is to press **PAUSE** (one of the rectangular keys above the typewriter part of the keyboard). When you press this key, whatever is on your screen at that moment will “freeze”. This program interruption, however, is only temporary. If you press **PAUSE** once again, your program will continue to run as before. So the **PAUSE** key really has two functions: pressing it once interrupts a program; pressing it again causes program execution to resume. (This kind of switch is often referred to as a “toggle switch”.)

You're already familiar with the second method of interrupting program execution — the key combination **SHIFT** **BREAK**. This is the uppercase function of the **PAUSE/BREAK** key. Whereas **PAUSE** causes a temporary program interruption, **SHIFT** **BREAK** terminates program execution and returns your Model 100 to the BASIC Command Mode. Try it. You'll get a message on your screen similar to this:

```
Break in 20
OK
```

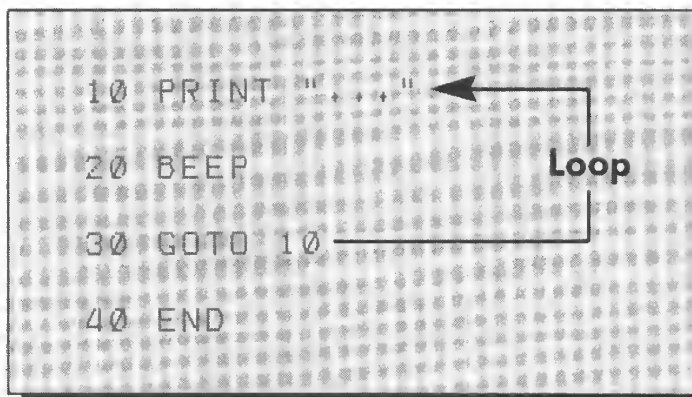


Figure 8-2. An endless loop

We said *similar* because the number might be different: it could be any one of the line numbers within the loop. That we got “20” means that the instant we hit **BREAK**, the program was executing line 20. Note that the Ok prompt is back on the screen, which is how your Model 100 lets you know that it’s ready to receive further BASIC instructions.

The FOR...NEXT...STEP Loop

Enough of endless loops. Let’s look at one of BASIC’s true gems: a pair of statements called the FOR statement and the NEXT statement, to which we’ll refer jointly as FOR...NEXT. This pair of statements provides us with a direct and powerful way to produce *controlled* loops. Let’s write a simple program that will help reveal the inner secrets of the FOR...NEXT statements:

```
10 FOR J = 1 TO 4
20   PRINT J
30 NEXT J
40 PRINT "done!"
50 END
Ok
RUN
1
2
3
4
done!
Ok
```

You can see that the single PRINT statement in line 20 printed all four numbers. This happened because PRINT in line 20 is part of a loop that was executed exactly four times. Only after all the looping was done did line 40 print “done!”. The statement FOR J = 1 TO 4 in line 10 defines the beginning of the loop, and the statement NEXT J defines the end of the loop. Whatever is between the FOR and NEXT statements is called the *body* of the loop — in this example that’s just the single PRINT statement. We indented the word *PRINT* to help us visually identify the body of the loop.

However, FOR...NEXT statements do more than define the loop’s boundaries; they also control the loop. To see how this works, let’s follow the execution of this program. The first statement to be executed is FOR J = 1 TO 4 in line 10. This tells the computer to assign a value 1 to a variable J, usually called the *index* or *counter* variable. It also causes the computer to decide whether to proceed with the loop, that is, whether to go on to the *PRINT* statement in line 20, the body of the loop. If the present value of J

is less than or equal to 4 (the number listed after TO), the body of the loop is executed. But if J has a value larger than 4, the loop is terminated and program execution proceeds with line 40 (the line immediately after the NEXT statement). Since the value of J at this stage is 1, which is clearly less than 4, the loop is permitted — that is, execution proceeds with the body of the loop (the PRINT statement in line 20).

Line 20 prints the value of J, which is 1. That's why we wrote PRINT J into the body of our loop: it tells us what the value of J is each time that statement is executed. Following line 20, the statement NEXT J is executed. It does just what it suggests: it determines the *next value of J* by adding 1 to the value passed down through the loop. Because the old value of J is 1, the "NEXT" value is 2. At this point, NEXT directs program execution back to the FOR statement — and we begin the loop all over again at line 10.

The second time through the loop, the statement FOR J = 1 TO 4 tests the value of J to see if it's equal to or less than 4. Because the present value of J is 2, which is less than 4, execution resumes with PRINT J, which prints the number 2 — the second number in our output. NEXT J again increments J by 1 to give 3 and passes program control back up to FOR J = 1 TO 4. And so it goes ... *until* we get to the fourth cycle of this loop.

At that point, PRINT J returns 4, NEXT J increments the value of J to 5, and FOR J = 1 TO 4 makes its test. This time, however, the test fails: J has the value 5, which is clearly *larger* than 4, and consequently the loop will *not* be executed again. Instead, execution "drops" through the loop to resume with the first statement *after* NEXT J, which in our example prints "done!". Whew! It takes a lot longer for us to "loop" through these explanations than it takes your Model 100 to execute the actual loop. Figure 8-3 shows how we can visualize this whole looping process.

In our previous example, the loop proceeded as long as J had a value from 1 to 4. We got one loop for each acceptable value of J — four loops in our example. So it's easy to see how we can change the number of loops to, say, 7 by replacing the number 4 with a 7 in the FOR statement.

More About the Index Variable

The FOR...NEXT statements are actually much more flexible than you might infer from the above example. For one thing, the beginning value of J doesn't have to be 1: it can be any integer, positive or negative. (It can

even be a decimal number, although that's generally not too useful.) Try this program:

```
10 FOR J = -3 TO 2
20 PRINT J;
30 NEXT
40 END
OK
RUN
-3 -2 -1 0 1 2
OK
```

This time *J* starts with a value of -3 , and the last loop executed has a counter value of 2 . There are six acceptable values of *J* — -3 , -2 , -1 , 0 , 1 , and 2 — so six loops will be executed. Later in the chapter we'll provide an example in which it is useful to start the counter with a value other than 1 .

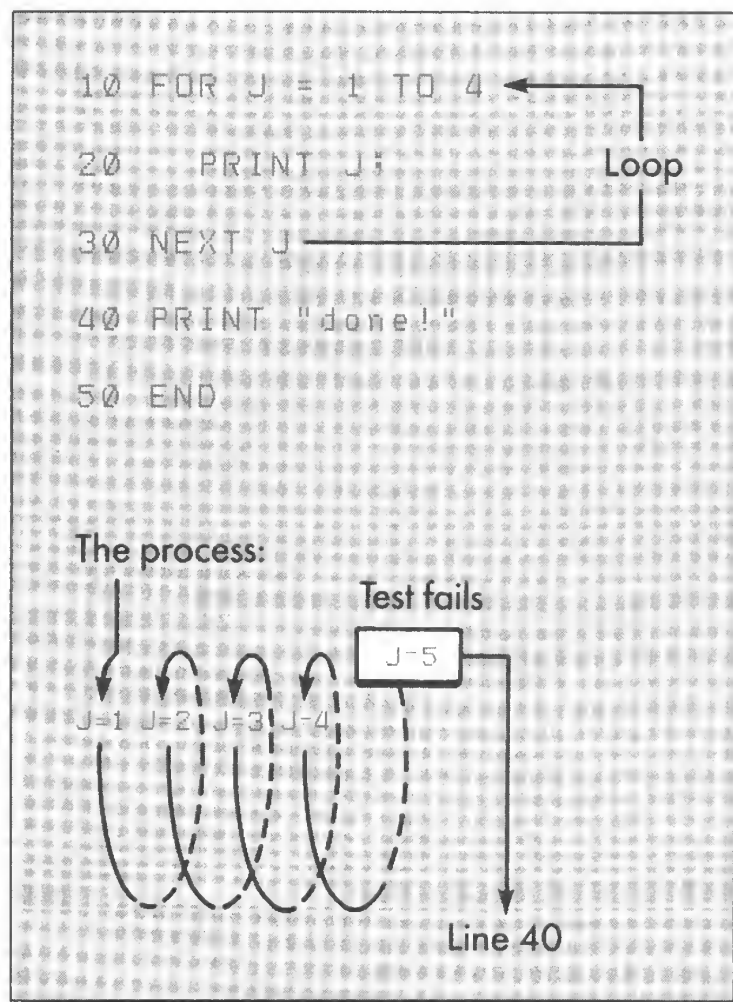


Figure 8-3. The FOR...NEXT Loop

Undoubtedly you've noticed that we sneaked in a few other changes. First, the output is displayed horizontally. That happened because we put a semicolon at the end of PRINT J in line 20. Remember that a semicolon at the end of a PRINT statement always suppresses the carriage return so that the *next* PRINT will do its job on the *same* line. The second change is more subtle — can you find it? We left out the J in the NEXT statement! Yes, it is perfectly legal to leave out the counter variable name in a NEXT statement. Some people even prefer it because it speeds up the program — very slightly to be sure but perhaps significantly in especially large programs.

The FOR...NEXT Statement

The following is an example of the simplest form of the FOR...NEXT loop:

```
10 FOR J = 3 TO 13
*
*
*
90 NEXT J
```

Initial value of J

Final value of J, the index or counter variable

Index variable is optional here

The FOR statement initializes the *index* or *counter* variable to the *initial value*, the number 3 in this example. The NEXT statement increments the value of J by 1. The loop will be executed as long as the index variable J has a value less than or equal to the final value — 13 in this example.

Variables as Index Limits

We can also use *variables* (with values assigned to them) to determine the initial and final values of the index variable. This feature allows us to

INPUT the range of the index variable, as illustrated by the following program:

```
10 INPUT "Beginning value-"; BEGIN
20 INPUT "Final value-----"; FINAL
30 FOR J = BEGIN TO FINAL
40   SQUARE = J * J
50   PRINT J, SQUARE
60 NEXT
70 END
OK
RUN
Beginning value-? 5
Final value-----? 9
 5          25
 6          36
 7          49
 8          64
 9          81
OK
```

The variables `BEGIN` and `FINAL` determine the smallest and largest values of the index variable. The fact that these are variables gives this program a flexibility it wouldn't have if we used fixed numbers.

We've done something else in this program to illustrate the use of the `FOR...NEXT` loop: we've made a table of numbers and their squares! The `FOR...NEXT` loop is ideal for generating tables of all kinds. We'll give some more sophisticated examples later in this chapter.

Two Examples Using the `FOR...NEXT` Loop

The `FOR...NEXT` loop opens up an almost infinite number of exciting programming possibilities. Before we go on to our next topic, we'd like to show you two different examples that illustrate the power and utility of the `FOR...NEXT` loop.

The first example is a program that draws a very simple graph of the square of a number. We'll use `TAB` to position an asterisk so that its column

position is equal to the square of J, our index. For the sake of simplicity, we won't use INPUT this time. Here's our program:

```
10 FOR J = 1 TO 5
20   SQUARE = J * J
30   PRINT TAB(SQUARE) "*"
40 NEXT
50 END
RUN
```

*

 *

 *

 *

 *

OK

Your first graph! Notice the slight curve in the line of asterisks. SQUARE gets large really fast! As you can see from line 20, SQUARE is assigned the value of the product of J times J — the square of J. Line 30 causes the asterisk to be PRINTed in the column equal to the value of SQUARE. So the bigger the value of SQUARE, the farther to the right is the asterisk.

Our second example may have some relevance to your financial affairs. Suppose you invest a certain amount of money in an account that earns a fixed yearly interest; the interest, however, is to be compounded monthly. The following program prints out a table that shows the month and present value of your investment:

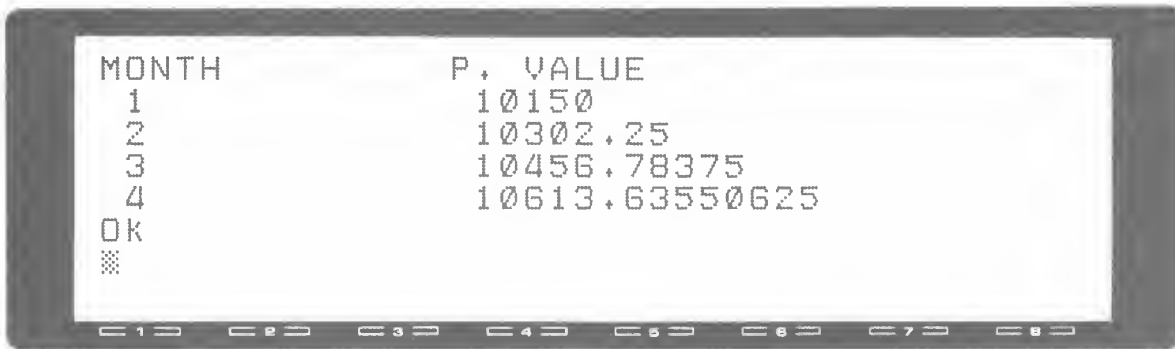
```
100 REM--NAME: "INVEST1"-----
110 REM
140 INPUT "Interest rate (%)--"; IR
150 INPUT "Amount invested-----"; PV
160 INPUT "Invest. period (mo)-"; LAST
170 REM
180 CLS
190 PRINT "MONTH", "P. VALUE"
200 PRINT "-----", "-----"
210 FOR MO = 1 TO LAST
220   IN = PV*IR/1200 ← Interest
230   PV = PV + IN    ← Updates Present Value
240   PRINT MO, PV
250 NEXT
260 END
RUN
```

Interest rate (%)--? 18

Amount invested-----? 10000

Invest. period (mo)-? 4

Immediately after the last entry (4), your screen clears and the following results appear:



MONTH	P. VALUE
1	10150
2	10302.25
3	10456.78375
4	10613.63550625

OK
␣

F1 F2 F3 F4 F5 F6 F7 F8

This is a pretty nice investment! If you enter a value for the investment period of more than four months, your program output would scroll upward so that the first figures would disappear. Remember, though, that you can always use **PAUSE** to “freeze” your scrolling output so that you can read it. Also, if you have a printer, you can get a hard-copy output of your program by using LPRINT in place of PRINT.

Most statements in this program are old hat for us now, but the body of the FOR...NEXT loop requires some explanation. Line 220 calculates the monthly interest earned on your investment (IN) by multiplying the present value (PV) by the monthly decimal interest rate, which equals the interest rate (IR) divided by 1,200. Why 1,200? Because interest is entered as a percentage, we first divide the interest rate (IR) by 100 to give us the decimal value. Then, because the entered interest represents the *yearly* rather than the *monthly* interest rate, we again divide IR, this time by the number of months in a year (12). Dividing first by 100 and then by 12 is the same as dividing by 1,200.

The next line (230) may seem a bit puzzling at first because the variable PV appears on both sides of the equal sign. Remember, however, that an equal sign in BASIC means “assign”, not “equal”. That is, $PV = PV + IN$ is not an algebraic equation; it is an instruction to find the sum of the present values of PV and IN and then assign that sum to the variable PV. With every execution of the loop, the variable PV is updated: we start with an *old* value of PV and end up with a *new* value of PV. Now we’re getting into some pretty sophisticated and useful programming!

Adding STEP to the FOR...NEXT Loop

Let’s explore some variations of the FOR...NEXT loop. The FOR...NEXT statements described so far are already pretty powerful. But there’s an

optional feature that adds even more flexibility: we can change the *interval* between successive index values by appending the FOR statement with the word *STEP*, followed by the size of the desired increment. The following program illustrates this expanded version of the FOR...NEXT loop:

```

10 FOR L = -4 TO 4 STEP 2
20   PRINT L;
30 NEXT
40 END
OK
RUN
-4 -2  0  2  4
OK

```

← 'STEP 2' indicates that L should be incremented by 2 with each execution of the loop

Line 10 assigns the initial value of -4 to the index L (we used L here so that you wouldn't suppose that the index must always be named J). The output shows that each time the loop is executed, the number 2 — not 1 — is added to the previous value of L. Of course "STEP 2" is responsible for this phenomenon. STEP 2 tells the computer to increment the index by 2 with each execution of the loop.

The number following STEP can be any integer, even a negative one. Try this variation of the above program:

```

10 FOR L = 5 TO -8 STEP -2
20   PRINT L;
30 NEXT
40 END
OK
RUN
5 3 1 -1 -3 -5 -7
OK

```

STEP -2 causes the number -2 to be added to L each time the loop is executed (adding -2 is the same thing as subtracting 2). We start with 5 and add -2 each time NEXT is executed. The last value of L to be printed is -7 . The "NEXT" value of L, -9 , is *not* printed because -9 is *smaller* than -8 , the final value of the index. When the program steps in the negative direction, the loop continues to be executed as long as the index is equal to or *larger* than the final index value — that is, as long as the index hasn't passed through the final value. Figure 8-4 may help clarify this stepping procedure.

The following program shows how STEP can be used in a somewhat more interesting way. Suppose you want to write a program that gives the

sum of all even numbers between, say, 0 and some final number. Here's how you might do it:

```
10 INPUT "Last number to be summed";LAST
20 SUM = 0
30 FOR N = 2 TO LAST STEP 2
40   SUM = SUM + N
50 NEXT N
60 PRINT SUM
70 END
RUN
Last number to be summed? 100
2550
OK
```

This is what happens. Line 10 asks for the value of LAST, the last number to be summed. Line 20 *initializes* the variable SUM to equal 0 (actually, it isn't essential that we do this, because the default value of a variable is 0; for the sake of clarity, however, it is a good practice to be explicit about this sort of thing). Line 30 begins the loop with N, the index, initially set to 2. Line 40 finds the sum of the present values of SUM (=0) and N (=2) to give the value 2, then assigns this value to the variable SUM. (Again, remember that the equal sign means "assign", not "equal".) Line 50 increments N by 2 and the loop starts over again.

Now the value of N is 4, which in line 40 is added to SUM. Each time the loop is executed, the value of N is added to SUM until we add our LAST value of N. Line 60 then prints the last value of SUM. Well, that's just what we wanted — the sum of all even numbers between 0 and LAST.

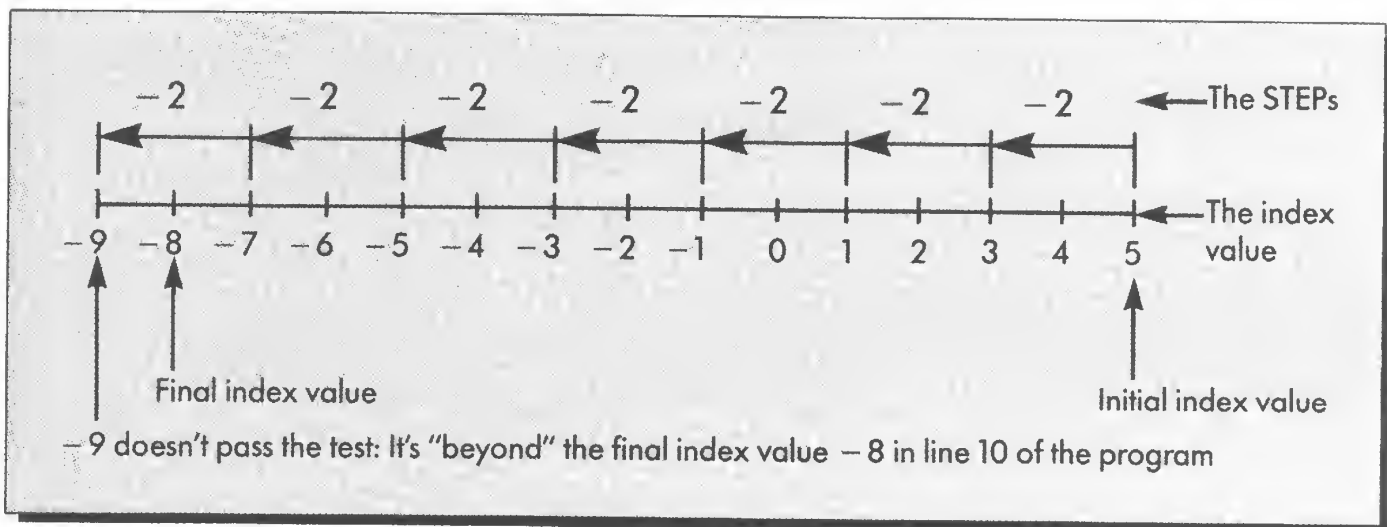


Figure 8-4. "STEPing" with the negative number -2

It's time for a grand summary of the FOR...NEXT statements.

The FOR...NEXT Statement with the STEP Option

The example shown below illustrates the complete syntax for the FOR...NEXT...STEP statement:

The diagram shows the following code snippet with annotations:

```
10 FOR M = -4 TO 10 STEP 2
*
*
*
*
80 NEXT M
```

Annotations with arrows pointing to the code:

- "The 'counter' or 'index' variable" points to **M**.
- "Amount that M is incremented with each loop" points to **2**.
- "Initial value of M" points to **-4**.
- "Final value of M" points to **10**.
- "Listing of index M is optional" points to **M** in the **NEXT** statement.

The FOR...NEXT statement defines the boundaries of a loop. FOR initializes the index to the initial value, then determines if another loop is to be executed by comparing the present index with the final value of the index. The loop is executed as long as the value of the index or counter variable (M in our example) is equal to or between the initial value (-4) and the final value (10). NEXT increments the present value of the index by the value listed after STEP (2 in our example).

Nested FOR...NEXT Loops

Some programming applications require one FOR...NEXT loop to be placed *inside* another FOR...NEXT loop. To see how such *nested* loops work, try the following program:

```
10 FOR J = 1 TO 2
20   PRINT "Outer loop; index ="; J
30   FOR K = 1 TO 3
40     PRINT "  Inner loop; index ="; K
50   NEXT K
60 NEXT J
70 END
```

```

RUN
Outer loop; index = 1
  Inner loop; index = 1
  Inner loop; index = 2
  Inner loop; index = 3
Outer loop; index = 2
  Inner loop; index = 1
  Inner loop; index = 2
  Inner loop; index = 3
OK

```

The output shows how the program works. Line 20 prints “Outer loop; index = 1” for the first loop (index $J = 1$), which is defined by the FOR...NEXT statements in lines 10 and 60. Then the inner loop, defined by the FOR...NEXT statements in lines 30 and 50, is executed three times for index values $K = 1, 2$, and 3. With each execution of this loop, the PRINT statement in line 40 prints out “Inner loop; index =” and the value of the loop index K . The whole process is then repeated once again for the second execution of the outer loop ($J = 2$), defined by lines 10 and 60.

To summarize, the inside or *nested* FOR...NEXT loop is completely executed (for all values of the index of the nested loop) for each value of the index of the *outer* loop. Not so hard, really.

Let’s use nested loops to “fill out” the graph we made earlier that shows the squares of a series of integers. Try this program:

```

10 FOR N = 1 TO 6
30   SQUARE = N * N
40   FOR J = 1 TO SQUARE
50     PRINT "*"
60   NEXT J
70   PRINT
80 NEXT N
90 END
OK
RUN
*
****
*****
*****
*****
*****
*****
*****
*****
*****
*****
OK

```

Pretty fancy stuff! The inside loop simply draws each horizontal line of asterisks. The length of each line is determined by SQUARE, which is equal

to the square of the index N of the *outside* loop. The PRINT statement in line 70 causes a carriage return after each line of asterisks is drawn (with this statement, the six lines of asterisks would be drawn on a single but folded line). The outside loop evaluates SQUARE and causes the inside loop to be executed six times. That's how we get six lines. (Nested loops may seem a bit complicated at first, but the technique is a powerful one. Such loops are also very useful in applications we'll explore later on. One example is provided in the exercises at the end of this chapter.)

Letting the Computer Make Decisions

Making decisions is often as important in a computer program as it is in your own life. Many personal decisions take the general form that *if* something is true, *then* we'll do so and so. For example, as we mentioned earlier, you might say

If the sun is shining then I'll wear my shorts

BASIC conveniently provides you with a way to translate such a statement into something your Model 100 can understand:

```
10 INPUT "is the sun shining "; ANS$
20 IF ANS$ = "yes" THEN PRINT "Wear your shorts"
30 END
RUN
is the sun shining? yes
Wear your shorts
OK
```

If you RUN this program again but answer "no", the program will just ignore you:

```
RUN
is the sun shining? no
OK
```

The output depends on the decision made by line 20, which says: IF "yes" has been assigned to the variable ANS\$, THEN go ahead and PRINT "Wear your shorts". If the string "no" or *any* string other than "yes" has been assigned to ANS\$, the IF...THEN statement does nothing. Program execution then continues with the next line.

The general form of IF...THEN looks like:

```
70 IF expression THEN clause
```

where the word *expression* refers to some kind of condition on which the decision depends. In our “shorts” example, this expression was an equality involving a string variable (ANS\$) and a string constant (“yes”). The word *clause* refers to the particular instruction to be carried out IF the expression is true. In our example, the clause was PRINT “Wear your shorts”. Both the expression and the clause can take a vast variety of forms. We’ll investigate these later in this section.

Using ELSE to Clarify Options

In our previous example, IF...THEN caused “wear your shorts” to be printed IF ANS\$ = “yes” (the sun is shining). There is no response to any answer but “yes”. What if you wanted a response to the answer “no” as well? Does BASIC provide a simple way of saying something like the following?

*If the sun shines then wear your shorts
otherwise wear your raincoat.*

Yes, it does. To translate the above into a BASIC statement, we need only substitute the word *ELSE* for the word *otherwise*. Let’s modify the program above to include this option:

```
10 INPUT "is the sun shining "; ANS$
20 IF ANS$="yes"
    THEN PRINT "wear your shorts"
    ELSE PRINT "wear your raincoat"
30 END
```

Though we divided the IF...THEN...ELSE statement in line 20 into three separate lines (by typing spaces until the cursor “wrapped around” to the next line), we can certainly write it as one continuous statement that would fold over to form two lines on your Model 100 screen. We’ve broken up the line in the way we have to visually organize this rather long BASIC statement into the logically distinct phrases starting with the words *IF*, *THEN*, and *ELSE*. However, there is a disadvantage in using this format: the added spaces require more memory and slow down the execution of the program. So if you’re more interested in speed and conserving memory than in visual clarity of program lines, you’ll want to write IF...THEN...ELSE statements as one continuous line.

Now RUN the above program:

```
RUN
is the sun shining? yes
wear your shorts
OK
RUN
is the sun shining? no
wear your raincoat
OK
```

From the second RUN you can see that when ANS\$ is *not* equal to "yes", the PRINT statement following ELSE is executed. In general, the clause following ELSE is executed *only* if the condition following IF is *not* true. Not too hard, is it? The IF...THEN...ELSE statement reads just like normal English! Figure 8-5 summarizes how such *conditional* branching is executed.

The IF...THEN...ELSE Statement

The complete syntax of the IF...THEN...ELSE statement is

```
110 IF expression THEN clause ELSE clause
```

as in the following example:

```
40 IF ANS$="yes"
    THEN PRINT "go for it"
    ELSE PRINT "well, think about it"
```

The clause after THEN is executed if the expression following IF is *true*; the clause after ELSE is executed if the expression following IF is *false*.

Relational Operators

In our previous example, the decision made by the IF...THEN...ELSE statement was based on the *equality* of a string variable and a string constant. However, there are many other types of conditions to consider. For one thing, expressions can involve *numeric* values and variables in addition to *string* values and variables. Also, equality is just one among many possible

conditions — called *relational operators* — that the IF...THEN...ELSE statement can test. The following summarizes the use of these relational operators:

BASIC Symbol	Meaning	Example
=	Equals (not assignment!)	a = b + 2
<	Less than	x < 5
>	Greater than	MNTH > 12
<= or =<	Equal to or less than	a * b <= 20
>= or =>	Equal to or greater than	AMT >= SAL
<> or ><	Not equal to	5 <> 9

As you can see, numbers can be compared with each other in all possible ways. Note the equal sign here really means “equal” (as in $5 = 4 + 1$) and not “assign” (as in “assign” the sum of the value of X and 1 to the variable X, which in BASIC is written as $X = X + 1$). Also, note from the examples given that expressions can involve numbers and/or variables.

The following example shows how some of these relational operators might be used in a program:

```

10 INPUT "What time is it (AM)?";TIME
20 IF TIME < 7
    THEN PRINT "there's still time to snuggle"
30 IF TIME = 7
    THEN PRINT "get up, you lazy bum!"
40 IF TIME > 24
    THEN PRINT "your clock is kaput"; GOTO 60

```

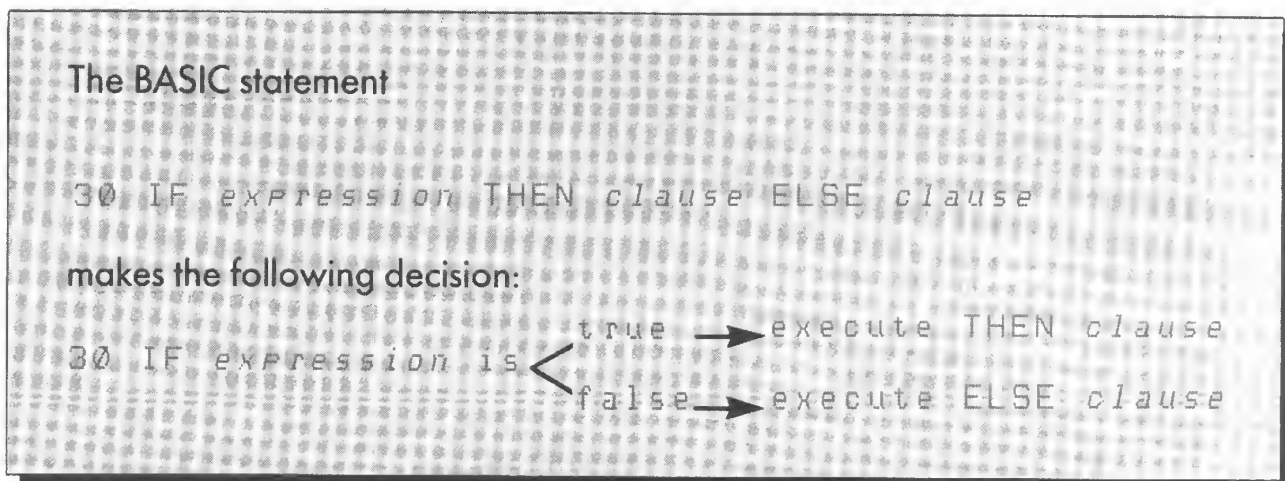


Figure 8-5. Decisions with the IF...THEN...ELSE statement

```

50 IF TIME > 7
    THEN PRINT "you're LATE, no time for coffee!"
60 END
OK
RUN
What time is it (AM)? 4
there's still time to snuggle
OK

```

This program is pretty self-explanatory, so we needn't run through all the possibilities. There's one new feature, though, which may have escaped your notice — line 40 contains *two* statements: (1) IF..THEN and (2), separated by a *colon*, GOTO 60, which causes END to be executed. A colon in a program line always separates the line into two distinct BASIC statements. Colons can be used either to speed up program execution by putting many BASIC statements into one program line or, as in our example, to simplify the program. The purpose of GOTO 60 in line 40 is to cause execution to bypass the next IF..THEN statement in line 50, provided that TIME > 24. That is, we don't want line 50 to be executed if the expression following IF in line 40 is true. If the expression in line 40 is *not* true, then the second statement (GOTO 60) will not be executed. Such stacked clauses allow one IF..THEN statement to control several functions. One decision can have many consequences.

Logical Operators Within the IF..THEN Statement

We can also write an IF..THEN statement to test for *two* conditions simultaneously. For example, the statement

```

50 IF WEATHER$ = "sunny" AND MONEY > 10000
    THEN VACATION$ = "Tahiti"

```

means the following: only if *both* WEATHER\$ = "sunny" *and* MONEY is greater than \$10,000 is the variable VACATION\$ to be assigned the value "Tahiti". That is, the whole expression is true only if *both* parts of the expression are true. The BASIC word *AND* is an example of a *logical operator*.

Two other examples of logical operators are XOR and OR. For example, the statement

```

35 IF WEATHER$ = "sunny" OR TEMP$ = "warm"
    THEN PRINT "wear bathing suit"

```

will print the phrase “wear bathing suit” if either one or both of the expressions are true — that is, if WEATHER\$ = “sunny” and/or TEMP\$ = “warm”. On the other hand, the operator XOR returns a true value only if *either one* of the listed expressions is true, but not if both are true.

The following example is useful in cases in which you’re not sure if a user will respond to an INPUT question with a “yes” or a “y” answer:

```
50 IF ANS$ = "yes" OR ANS$ = "y"  
   THEN PRINT "come closer"
```

This statement will cause the THEN phrase to be executed if the value of ANS\$ is “yes” and/or “y”. Because the variable ANS\$ can’t have both the values “yes” and “y” at the same time, it makes no difference here whether the operator XOR or the operator OR is used.

The Logical Operators OR, XOR, and AND

Given the expressions *A* and *B*, the operators OR, XOR, and AND have the following effects:

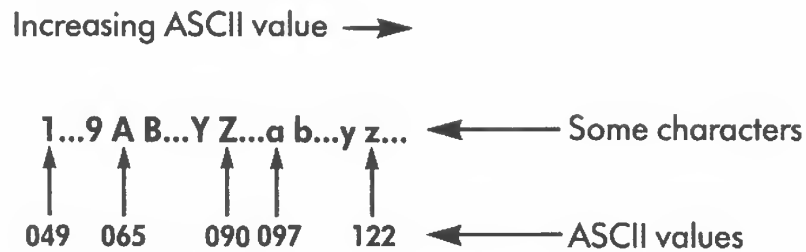
Combination	True Only If
<i>A</i> OR <i>B</i>	<i>A</i> and/or <i>B</i> is true
<i>A</i> XOR <i>B</i>	<i>A</i> or <i>B</i> is true, but not both
<i>A</i> AND <i>B</i>	Both <i>A</i> and <i>B</i> are true

Logical Operations on Strings

We now know that numbers can be compared in many different ways, but you may be surprised to learn that *strings* can be compared with each other in the same way. Stating that one string equals another makes sense. But what does it mean to say that “a” < “m”?

Well, computer people have attached a unique number to each character on your keyboard; and these combinations are given by *ASCII codes* which are listed in Appendix B. For example, the ASCII code for the letter *a* is 097; for the letter *m*, it is 109. Letters, and characters in general, are ordered by your computer by means of their ASCII code. It is in this sense that “a”

< "m". If you look at this code in Appendix B, you'll see that characters follow an order, as shown below:



If strings longer than one character are compared, the rule is that the first character of each string is compared, then the second character, and so forth.

Below are some examples of true statements:

```
"a" > "A"
"a" > "5"
"B" < "a"
"ba" > "bA"
"bb" > "ba"
"m r " > "mr "
```

← A space has an ASCII value of 032

Comparing strings in this manner is essential to programs that arrange words in alphabetical order.

More About the Clause in the IF...THEN Clause

The part following THEN or ELSE, which we call the *clause*, can be any legitimate BASIC statement. Here are some examples:

IF,,,THEN PRINT ".,.,."	
IF,,,THEN GOTO 300	← A conditional GOTO
IF,,,THEN 300	← Equivalent to the above
IF,,,THEN Y = X - 5	← Assignment statement
IF,,,THEN INPUT DAY	← Asks for user INPUT to assign DAY
IF,,,THEN END	← Terminates program
IF,,,THEN X=Y:GOTO 60	← Assigns Y to X and executes GOTO 60

Here's an example of a complicated but legitimate statement:

```
50 IF AMT > 4000 .AND INT >= .05
   THEN TOTAL = X + AMT: GOTO 4300
   ELSE PRINT "not enough"
```

Summary

In this chapter we have introduced several of the most important programming tools in BASIC. One of these tools is the *loop*, which causes one part of a program — the *body* of the loop — to be executed over and over again. An *endless loop* can be created by using a GOTO statement, which causes program execution to *branch* to a specified line number. A more sophisticated and usually more useful method of creating a loop is by means of the FOR...NEXT statement. In contrast to the GOTO loop, the number of loops executed by the FOR...NEXT statement can be controlled by specifying the beginning and final values of the *index* or *counter* variable.

Another powerful programming tool we introduced in this chapter is the IF...THEN...ELSE statement. This statement enables a program to make a decision in the form “If a certain condition or expression is true, then perform instruction A; otherwise, perform instruction B”.

Exercises

1. Write a program that prints out a table of temperature conversions from degrees Fahrenheit to degrees Centigrade in the range of 60 to 90 degrees Fahrenheit.
2. Modify the program “INVST1” shown in the text so that the output shows the value of your investment every *year*, not every month.

Solutions

1. The following program converts temperatures from degrees Fahrenheit to degrees Centigrade:

```
10 REM--NAME:"F->C"-----
20 REM
30 PRINT "DEG, F", "DEG, C"
40 PRINT "-----" "-----"
50 REM
60 FOR F = 60 TO 90
70   C = (F - 32)*5/9
80   PRINT F,C
90 NEXT F
100 END
```

```

RUN
DEG, F      DEG, C
-----
60          15.55555555555556
61          16.11111111111111
62          16.66666666666666
63          17.22222222222222
*           *
*           *
*           *
Ok

```

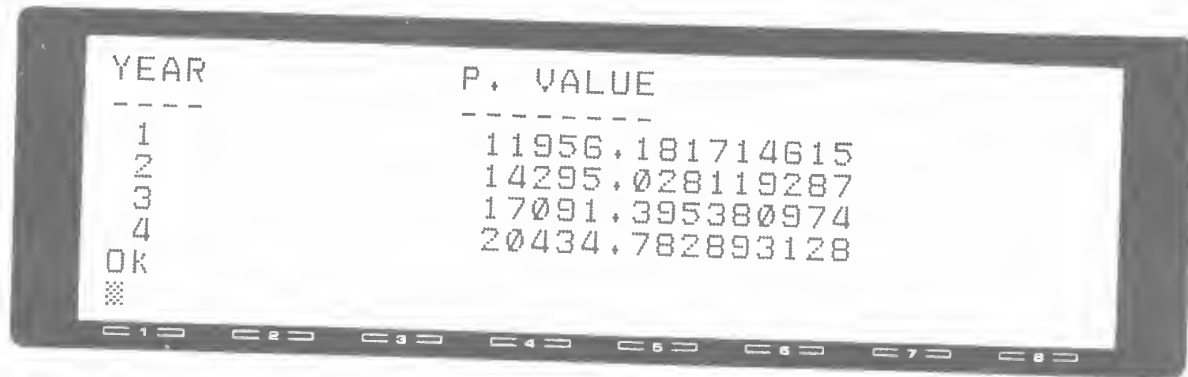
2. This is our modification of the program "INVST1", using nested FOR...NEXT loops:

```

100 REM--NAME: "INVST2"-----
110 REM
140 INPUT "Interest rate (%)----"; IR
150 INPUT "Amount invested-----"; PV
160 INPUT "Invest, period (yrs)-"; LAST
170 REM
180 CLS
190 PRINT "YEAR", "P, VALUE"
200 PRINT "----", "-----"
210 REM
220 FOR YEAR = 1 TO LAST
230   FOR MO = 1 TO 12
240     IN = PV*IR/1200
250     PV = PV + IN
260   NEXT MO
270   PRINT YEAR, PV
280 NEXT YEAR
290 END
RUN
Interest rate (%)----? 18
Amount invested-----? 10000
Invest, period (yrs)-? 4

```

As soon as the last value is entered, the following output appears on the screen:



YEAR	P. VALUE
----	-----
1	11956.181714615
2	14295.028119287
3	17091.395380974
4	20434.782893128
OK	
⊠	

Basically, our original loop — now the nested loop — is still the same, except that it compounds the interest for exactly twelve months (rather than a variable amount equal to the investment period), and it doesn't print PV any more. The new (outside) loop now prints the present value of your account (PV) at the end of each year. So the inside loop does all the calculating, and the outside loop prints the result after every twelve executions of the inside loop (one for each month of the year) — that is, at the end of each year.

This is a pretty sophisticated program. It wouldn't be hard to modify it to include monthly contributions or even the "front-end" loading typical of many investment programs. Here we have a powerful programming tool at our disposal.

9

Dealing with Numbers

Concepts

- Double- and single-precision numbers
- Integers
- Type declaration tags and statements
- Exponential and scientific notation
- Formatting numerical output

Type Declaration Tags #, !, %

Instructions

DEFSNG, DEFINT, DEFSGL, DEFDBL, DEFSTR, PRINT USING

Numbers represent quantities, such as distance, weight, or dollars. For example, a bathroom scale might reveal Brunhilde's weight to be 174 pounds. For most practical purposes, this three-digit number with no decimal or fractional part is a perfectly adequate representation of her weight. However, some applications require the use of both many digits and the decimal point. For example, a typical entry in the federal budget might be something like \$56,987,456.45. Many scientific calculations require the manipulation of *really* large (or small) numbers, such as 12,546,000,000,000,000,000 miles — the distance to our nearest spiral galaxy. Though most of us usually pay little attention to the different ways such numbers are represented, your computer makes distinctions between the kinds of numbers given in the examples above. In the first part of this chapter, we explore the various ways the Model 100 “thinks” about numbers — that is, about number *types*.

In the second part of the chapter, we deal with formatting numerical output with the PRINT USING statement. This statement enables us, for example, to print out neat columns of dollar amounts in a format that arranges all the decimal points in columns and adds the dollar sign (\$) as a prefix to the dollar amount.

Different Kinds of Numbers

The Model 100 divides numbers into three basically different types. These are called *double-precision* and *single precision numbers* (collectively called floating-point numbers) and *integers*. The difference between these types concerns the maximum size of the number, the number of digits “remembered” and manipulated, and the way the numbers are displayed. Each type has properties that makes it suitable for different applications. The following sections explain these three types of numbers, their differences, and the particular applications for which each is best suited.

Double-Precision Numbers

Whether you know it or not, the type of number we’ve been using all along has been the double precision number. For example, in Chapter 3 we showed the following example:

```
PRINT 2/3
      ,666666666666667
OK
```

This example gives the result of dividing 2 by 3 to be the fourteen-digit number .666666666666667. However, the *exact* decimal value of 2/3 would show an infinite number of sixes. Because it isn’t practical for the Model 100 to show an infinite number of digits, the people who wrote BASIC for the Model 100 decided to display fourteen digits — a number that is both practical to display and precise enough for almost any application.

The example above asked BASIC to print the result of a calculation, but, we can just as well ask BASIC to print out a fourteen-digit number that we type in:

```
PRINT 123,45678901234
      123,45678901234
OK
```

No problem — all fourteen digits that we entered after PRINT are accurately printed out.

All fourteen-digit numbers, such as those in our previous examples, are called *double-precision* numbers. The word *double* in double-precision will become more meaningful in contrast to *single-precision* numbers, which we'll discuss in the next section. Notice that we didn't do anything special to calculate and display a number to a precision of fourteen digits. Double-precision numbers and calculations are "natural" to the Model 100; we say that the double-precision number is the *default* number type on the Model 100. ("Default" refers to what you get if you don't specifically ask for something.)

You may wonder what happens when we deal with numbers that have fewer than fourteen digits: Are these also double-precision numbers? The answer is "yes". Consider the following example:

```
PRINT 10.1
10.1
OK
```

Though the number 10.1 only has three digits, your computer interprets this number as 10.100000000000; that is, if you enter a number like 10.1, the Model 100 automatically adds zeroes after the last digit to make it into a fourteen-digit number. This is also true when your computer prints out the result of a calculation, as in the following example:

```
PRINT 4/5
.8
OK
```

Although the quotient as printed out is the one-digit number .8, your computer actually calculates a fourteen-digit number — .80000000000000. The Model 100 doesn't show the long string of zeros after the 8; they are *implied*.

To summarize, the Model 100 automatically displays numbers and does calculations to a precision of fourteen digits.

Single-Precision Numbers

Many programming applications do not require numbers with a precision of fourteen digits. For example, suppose we invest \$24,550 at an annual interest of 10.22 percent. To find the monthly interest earned by this investment, we multiply 24,550 by the monthly fractional interest rate, which equals $10.22/1,200$ — (dividing by 1,200 is the same as first dividing by 12 (to get the monthly interest rate) and then dividing by 100 (to get the monthly fractional rate)). We might ask the Model 100 to perform this calculation in the following manner:

```
PRINT 24550 * 10.22/1200
209.08416666667
OK
```

Great, but we don't really care about a billionth of a penny! All we're interested in is the first part of the number — 209.08 — the rest is rather insignificant. We don't need double-precision numbers and calculations for this problem. For many programming applications, such as this example, numbers with only six digits suffice — numbers of a type called single-precision numbers. The following example illustrates how we can print a number in single-precision. Don't forget the exclamation point (!) after the number:

```
PRINT 1.2345678901234!
1.23457
OK
```

We typed in a fourteen-digit number, which BASIC would have ordinarily printed out as a fourteen-digit number; but this time BASIC returned only a 6 digit number! The new feature in our command — the exclamation mark immediately following the number — is responsible for causing BASIC to return the single-precision number 1.23457.

We can also append the exclamation mark to a variable name so that its value will always be interpreted as a single-digit precision number. Consider this example:

```
10 INPUT X!
20 PRINT X!
RUN
? -1.2345678901234
-1.23457
OK
```


The exclamation mark at the end of the variable X! means that the *value* of X! can only be a single-precision number, no matter what number is assigned to it. The variable X! is called a *single-precision variable*, and the exclamation mark (!) responsible for the type specification is called a *type declaration tag*.

What would happen in the program above if we forgot the exclamation mark after X in the PRINT statement in line 20? Nothing very serious, but whatever value we enter in response to INPUT, our program will always print a 0. You can guess the reason: the variables X! and X are *different* variables; therefore, no value will ever be assigned to X in the statement PRINT X. Consequently, the value of X will forever remain 0.

Now, consider the problem of printing a single-precision value that results from a calculation — say, the single precision value of 2/3. A statement such as 10 PRINT 2!/3! still returns a double-precision value. But try the following:

```
10 QUOTIENT! = 2/3
20 PRINT QUOTIENT!
RUN
,666667
OK
```

Despite the fact that the quotient 2/3 has a double-precision value, the variable QUOTIENT! accepts only the first six digits. The value of QUOTIENT! will always be a single-precision number.

Earlier we suggested that single-precision numbers are adequate for many applications. But you may wonder if there is any real advantage to single-precision over double-precision numbers that would make it worthwhile to declare numeric types. For relatively short programs, the answer is “no”. But for longer programs, working with single-precision rather than double-precision numbers can save a significant amount of memory space and speed up program execution. To be precise, whereas it takes only four bytes of memory to store a single-precision number, it takes eight bytes to store a double-precision number. (A *byte* is equivalent to the amount of memory required to store one character or one letter of the alphabet.)

Summary of Double- and Single-Precision Numbers

The table below illustrates some of the important differences between double- and single-precision numbers.

Number Type	Number of Significant Digits	Tag for Variable or Number	Memory Required
Double-precision	14	#(optional)	8 bytes
Single-precision	6	!	4 bytes

Unless otherwise specified, the Model 100 expresses all numbers in the double-precision mode (hence the # tag is usually not required).

The next table lists some examples of these two types of numbers and their associated variables:

Double-Precision Numbers and Variables	Single-Precision Numbers and Variables
AMOUNT	DISTANCE!
LARGE#	M1!
1.2345678901234	123.456!

The exact range in the size of double- and single-precision numbers is discussed in the section on scientific notation and is summarized in the box at the end of that section.

Scientific Notation — Exploring Large and Small Numbers

Very large and very small numbers are expressed by the Model 100 in *scientific notation*. Though this method of expressing numbers derives from scientific applications, we certainly needn't be scientists to use it. We will, however, assume that you already know what raising a number to a power means. If not, you can probably skip this section without harming your understanding of subsequent chapters of this book.

Scientific notation works the same way in both single- and double-precision numbers. Since double-precision is the "default" mode of the Model 100, most of our examples will use double-precision numbers.

Large Numbers

Nothing unusual happens when we ask PRINT to print out a double-precision integer that has up to fourteen digits. For example, let's print out the 1981 national debt:

```
PRINT 29377160000000
29377160000000
OK
```

But what happens if we exceed fourteen digits — the maximum number the Model 100 can print out? Or in single-precision, what happens if we try to print out a number larger than 999,999? The best way to find out is to try it. Let's enter a really large double-precision number, such as the distance in miles to our closest spiral galaxy (Andromeda) — a nineteen-digit number:

```
PRINT 12546000000000000000
1.2546E+19
OK
```

We entered a number in the usual manner, but PRINT returned the same number in *scientific notation*. As printed out on the Model 100,

1.2546E+19 means $1.2546 * 10^{19}$

in standard mathematical notation. The only difference between the scientific notation of the Model 100 and standard notation is the letter *E*, which means “raise 10 to the power specified by the following two-digit number”. The Model 100 always displays large numbers in scientific notation if the number to be printed has more than fourteen digits to the left of the decimal point (which in our example is assumed to be located after the last zero).

If you're a bit rusty in dealing with scientific notation, Figure 9-1 summarizes how a number written by the Model 100 in scientific notation can be converted to standard notation.

Small Numbers

So far we've talked only about large numbers. How are small numbers printed by the Model 100? To find out, try the following examples:

```
PRINT .0227
.0227
OK
PRINT .00227
2.27E-3
OK
```

In the first example, BASIC prints out the number .0227 just the way we've typed it. However, when we ask BASIC to print .00227, the result is 2.27E-3, which is the same number expressed in scientific notation. As printed out by the Model 100, the number

2.27E-3 means $2.27 * 10^{-3}$

in standard mathematical notation. BASIC on the Model 100 always displays numbers smaller than .01 in scientific notation.

The Largest and Smallest Numbers

Table 9-1 lists several examples of very large and very small numbers in both scientific and standard notation. As you can see, the Model 100 can represent extremely large and small numbers. But what are its limits? For all practical purposes, the answer is the same for single- and double-prec-

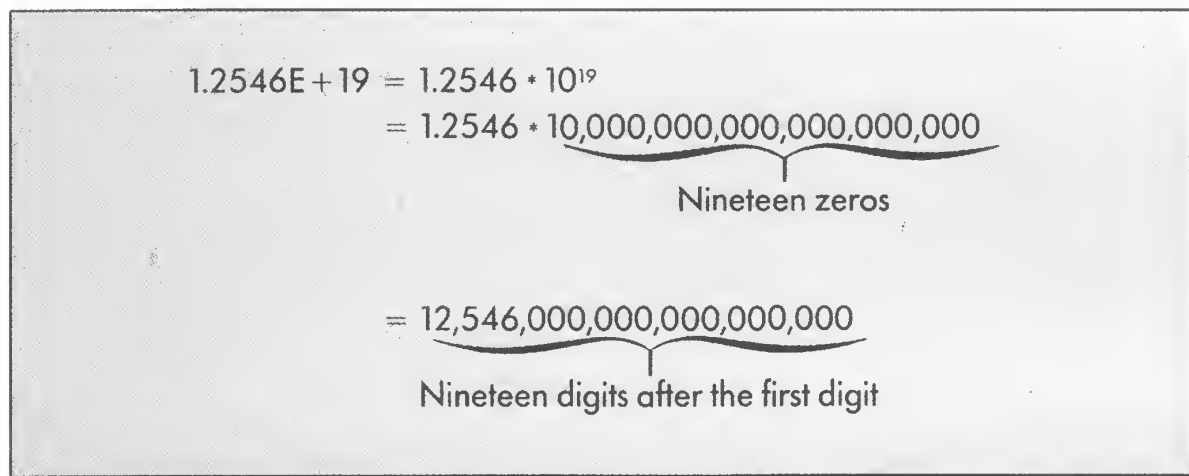


Figure 9-1. Changing a number in scientific notation to normal format

sion numbers. In scientific notation, the largest number the Model 100 can express has an exponent of 62; the smallest non-zero number has an exponent of -64 . Consequently, in single-precision, the largest and smallest numbers are the following:

largest number: $9.99999\text{E} + 62$

smallest number: $1\text{E} - 64$

In double-precision, we can add a few more nines, although these make only a very small difference. This range of numbers enables us to express on the Model 100 practically *any* quantity we can think of.

Summary of Scientific Notation

The number $5.678\text{E} + 5$ can be rewritten in the following manner:

$$\begin{aligned} 5.678\text{E} + 5 &= 5.678 * 10^5 \\ &= 5.678 * 100,000 \\ &= 567,800 \end{aligned}$$

The small number $5.678\text{E} - 5$ can be rewritten this way:

$$\begin{aligned} 5.678\text{E} - 5 &= 5.678 * 1/10^5 \\ &= 5.678 * .00001 \\ &= 0.00005678 \end{aligned}$$

The largest number that can be expressed by the Model 100 has an exponent of 62; the smallest, aside from zero, has an exponent of -64 .

Number in BASIC Scientific Notation	Number in Standard Notation	Comments
$3.5\text{E} + 2$	3500	
$1.3\text{E} + 17$	130,000,000,000,000,000	Age of earth in seconds
$2.997924590\text{E} + 8$	299,792,459.0	Speed of light in m/s
$5.0\text{E} - 11$.000 000 000 05	Radius of atom in m
$9.10953\text{E} - 28$.000 000 000 000 000- 000 000 000 000 910 953	Mass of electron in g

Table 9-1. Examples of numbers written in both scientific and standard notation

Integers — Saving Memory Space

Though almost any numerical quantity can be expressed perfectly well on the Model 100 as a double- or single-precision number, it is sometimes convenient and useful to express a quantity in terms of *integers* — whole numbers, with no fractional or decimal part. For example, the numbers 5, -45, and 35,000 are integers, while 3.2 and 1/3 are not. Though BASIC in double- or single-precision mode can print out numbers that *look* like integers, the Model 100 still treats such integers as double- or single-precision numbers. That is, the Model 100 thinks of the double-precision number 3 as 3.000000000000000 and in the process uses eight bytes of memory. Single-precision requires four bytes. Mainly to save memory space, and speed program execution, BASIC on the Model 100 allows us to declare a variable or number to be of the *integer type*. A number of the integer type has no decimal part and requires only two bytes of memory.

We can declare an integer type in the same way we declared a single-precision type, that is, by adding a *type tag* to the number or the variable that is to represent the number. The tag for integers is the percent sign (%). Consider the following example:

```
PRINT 6.66666666%  
6  
OK
```

The percent tag at the end of the decimal number caused BASIC to “remember” the number as an integer. Notice that the number 6.666666666 is *not* rounded off to 7; instead, the whole decimal part of the number is simply dropped or “truncated”.

The following example illustrates how variables can be declared as integer types and how integer arithmetic can be used:

```
10 INPUT "Two numbers"; X%, Y%  
20 Z% = X%/Y%  
30 PRINT X%; Y%; Z%  
RUN  
Two numbers? 4.7, 3.0111  
4 3 1  
OK
```

The first two numbers printed out, 4 and 3, are simply the integer parts of the numbers we entered, 4.7 and 3.0111, respectively. The third number printed isn't quite so obvious: it's the integer part of the quotient 4/3 (which is 1.333 ...).

There is a limit to the size of an integer number on the Model 100. The largest number we can represent as an integer type is 32,767. That's a large number but of course not nearly as large as $9E + 62$! If we need to represent integer quantities larger than 32,767, the best alternative is to use single precision numbers. Integers can also be negative; the "largest" negative integer number on the Model 100 is $-32,768$.

One application of integer variables is as the counter or index variable in a FOR...NEXT loop. The following is a simple example of a FOR...NEXT loop that uses an integer index variable:

```
10 AMOUNT = 1000
20 FOR MNTH% = 1 TO 6
30   PRINT MNTH%, AMOUNT
40   AMOUNT = AMOUNT*1.01
50 NEXT
RUN
1      1000
2      1010
3      1020.1
4      1030.301
5      1040.60401
6      1051.0100501
OK
```

This program prints out the monthly balance of an account that earns 12 percent annual interest compounded monthly. The variable MNTH% is our integer index variable. (Incidentally, we wanted to use the variable MONTH%, but an error statement reminded us that part of that variable, ON, is a reserved word!) The output of this program would be the same if we dropped the integer type tag after the variable MNTH%; but the program would then require more memory and would run slower. Though this consideration is unimportant in such a short program, it becomes significant in a longer program, especially if want to store it as a RAM file.

Summary of Integer Numbers and Variables

We can declare numbers and variables to be of the integer type by appending a percent sign (%) to the number or to the variable. Integer numbers require only two bytes of memory. The following are some examples of integer-type declarations:

40 X = 345.56%	← Assigns the integer 345 to the variable X
50 J% = 3.45	← J% has the value 3

Numbers expressed as integer types must lie in the range of $-32,768$ to $32,767$.

Type Declarations — DEFINT, DEFSNG, DEFDBL, DEFSTR

You now know how to specify a number or variable as any one of the three possible types by using type declaration tags (#, !, %). There is, however, another way to specify a variable type that is sometimes more convenient to use. Instead of using tags, you can use *variable declaration statements* at the beginning of a program to specify the type of any or all of the variables to be used in the program. With this method, you need only be concerned about type declarations once, at the beginning of the program. That way you don't have to bother with all the type declaration tags, which are easy to forget when writing a program!

The following statement illustrates how to declare all the variables that *begin* with the letters S, J, and M as integer variables:

```
10 DEFINT S, J, M
```

This BASIC word *DEFINT* suggests the purpose of this statement: it causes the variables defined by the listed letters after DEFINT to be “DEFined as INTegers”. The listed letters must be separated by commas. The letters themselves define all the variables that *begin* with those letters, which means that if you use the variables SLOP, MONEY, M, and JET in the remainder of the program, these variables will automatically be of the integer type — without the need to append the percent (%) tag to these variable names.

We can use *both* type declaration statements and type declaration tags in a single program. However, if there is any conflict between them, the declaration tag *overrides* the declaration statement. For example, if we use the

variable `MONEY#` after the declaration statement above (`10 DEFINT S, J, M`), the variable `MONEY#` would be interpreted as a double-precision variable, despite the declaration statement.

If our type declaration statement is to list adjacent letters in the alphabet, we can use the following shortcut: instead of listing all the letters, we simply list the first and last letter in the series, separated by a dash. For example, the statement

```
20 DEFINT A, L-N
```

defines all variables that begin with the letters A, L, M, and N to be integer variables.

The following examples illustrate the other type declarations using declaration statements:

<pre>20 DEFSNG X, A, D</pre>	← Declares all variables that start with the letters X, A, and D as single-precision variables
<pre>30 DEFDBL C, Q-Z</pre>	← Declares all variables that start with the letters C and all letters between Q and Z as double-precision variables
<pre>40 DEFSTR N, M</pre>	← Declares all variables that start with the letters N and M as string variables

The last statement in the examples above (using `DEFSTR`) has nothing to do with numbers; it defines a *string* variable. We included it here to make our discussion complete — we'll discuss strings in Chapter 15.

PRINT USING — Printing Numbers a Better Way

In this part of the chapter we focus on printing output with the `PRINT USING` statement. Because you already know how to print numbers and strings in various formats with the `PRINT` statement, you may wonder about the need for another `PRINT`-related statement. The answer is that although the `PRINT` statement gives us some control over the appearance of output, we often need to specify the format of program output much more precisely than we can with the `PRINT` statement. For example, we'd like to specify that the output of a dollar value show only two digits after the decimal point, that the decimal point is in a specific column position, and that a dollar sign appears before the dollar amount printed. The `PRINT USING` statement

gives us a tool with which to format this and many other types of numeric output. Though we can also use `PRINT USING` for printing out string values, we'll limit ourselves to printing numeric output, which is its principal application.

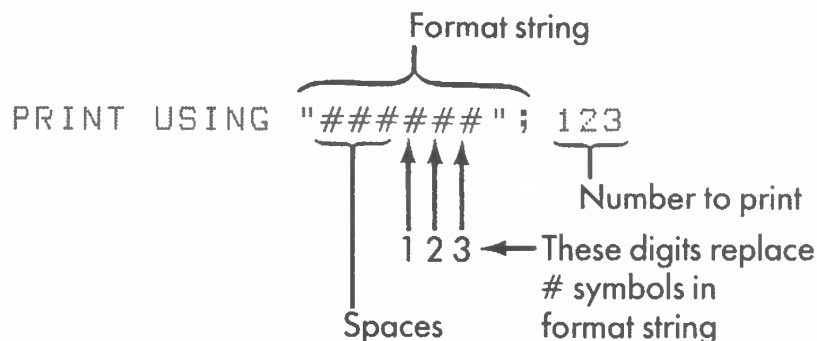
The Basic Format String of `PRINT USING`

The best introduction to `PRINT USING` is a simple example. Enter the following command:

```
PRINT USING "#####"; 123
123
OK
```

This command has two parts after the words *PRINT USING*, the second of which is the number that is actually printed out — 123 in our example. It is separated from the rest of the statement by a semicolon. The part between the word *USING* and the semicolon — the string “#####” — is called the *format string*. The format string, as its name suggests, specifies the format of the printed output. The number symbols (#) inside the format string are *place holders* for the digits in the number to be printed. When `PRINT USING` is executed, each digit of the number to be printed replaces a # symbol. As you can see from the output, the right-most digit replaces the right-most # symbol, resulting in what is called a *right-justified* number. When BASIC runs out of digits with which to replace the # symbols, it simply prints blanks. Because we specified six place holders in our example and our number had only three digits, `PRINT USING` printed out three blanks in front of the number 123.

Formatting Integer Output with the `PRINT USING` Statement



Our second example illustrates how we can use PRINT USING to print out a column of right-justified integers:

```
10 X = 9
20 FOR J = 1 TO 6
30   PRINT USING "#####"; X
40   X = X * 11
50 NEXT J
RUN
      9
     99
    1089
   11979
  131769
 %1449459
OK
```

The method used to generate this series of numbers isn't important; any procedure that generates rapidly growing numbers will do. What is important here is the PRINT USING statement in line 30. First, notice that we can use PRINT USING to print out the value of a variable (X in this example), just as we can use PRINT for this purpose. Second, the effect of the format string in PRINT USING is to print out *right-justified* columns; that is, the right-most digits in all but the last number line up in the same column position. If we had instead used a PRINT statement here, the output would have been *left justified*; that is, all the numbers would have been pushed over to form a neat column of the left-most digits of each number. The last number in the output, however, is too large to fit into the format string defined by the "#####". The Model 100 still manages to print out the whole number by letting it extend beyond the specified right-most column. In addition, a percent sign (%) printed in front of the number indicates that the number of digits we're printing exceeds the number of place holders.

The format string we've used in our examples consists of string constants, but we can just as well replace the format string constant with a format string *variable*. Consider the following program:

```
10 PRINT USING "#####"; 123
15 FMT$ = "#####
20 PRINT USING FMT$; 123
RUN
    123
    123
OK
```

Line 10 and lines 15 and 20 print the number 123 in the same way; the only difference is that the format string in line 20 is a string variable, while a string constant is used in line 10. Using string variables allows us to use a single PRINT USING statement to format output in different ways.

The PRINT USING statement also allows us to print out a *list* of values rather than just a single value as we've done before. For example, the command

```
PRINT USING "####"; 1,2,-3,4
1    2   -3    4
OK
```

uses the same format string over and over again to print out the listed values until all numbers are printed.

Different Format Strings

So far we've described a number of different ways to use PRINT USING to print out right-justified *integers*. However, we can also use PRINT USING to format *decimal* numbers in various ways and to add special characters, such as the dollar sign, to any number. In the following sections we introduce some formatting variations possible with PRINT USING.

Specifying the Decimal Point Position

We can effectively use PRINT USING to print out decimal numbers; we need only place the decimal point into the format string at the desired position. For example, let's write a statement that prints out the numbers 123.456789 and .123456789 to appear as typical dollar amounts:

```
10 PRINT USING "#####.##"; 123.456789
20 PRINT USING "#####.##"; .123456789
RUN
123.46
 0.12
OK
```

Great! Now we can print dollar amounts in a sensible manner. In the past, when our program calculated a number like 123.456789, PRINT always printed out the whole number, down to the thousandth or millionth of a penny. Notice also that the decimal points line up in the same column position, which is usually the preferred way to print out a column of dollar amounts. Line 20 also puts a zero to the left of the decimal point to give us the conventional (and preferred) output of 0.12 instead of .12.

Adding Commas to Numeric Output

Large numbers are easier to read if a comma is used to mark the position of every third digit, as in the number 45,000,000. The PRINT USING statement allows us to insert such commas in our printed numeric output. The following example illustrates how this is done:

```
10 PRINT USING "#####.##"; 1234567.8
20 PRINT USING "#####,.##"; 1234567.8
RUN
    1234567.80
  1,234,567.80
OK
```

Lines 10 and 20 are identical, except that line 20 has a comma immediately to the left of the decimal point in the format string. As you can see from the output, the effect of that single comma is to insert a comma in the output before every third digit to the left of the decimal point. Note that the comma in the format string also counts as a place holder; otherwise, the decimal points in the two numbers in the output would not be in the same column position.

Adding a Dollar Sign and Asterisks to Numbers

When printing out dollar amounts, we often want to include the dollar sign in front of the dollar amount. Once again, PRINT USING makes it easy to accomplish this task. Consider the following command:

```
PRINT USING "$$####.##"; 123.4567
  $123.46
OK
```

The code that causes a printed number to be preceded by a dollar sign is the double dollar sign (\$\$) in the first two characters in the format string. The blank space preceding the dollar sign in the output indicates that the two dollar signs in the format string are themselves place holders.

We can use a very similar code to change the leading blanks of a printed number into a row of asterisks. For example, the command

```
PRINT USING "#####"; 1234
*****1234
OK
```

prints out the number 1,234 with six leading asterisks. Again, the two asterisks in the format string count as place holders.

We can also print numbers with both the dollar sign and asterisks to fill the leading blanks. The required code in the format string is a double asterisk and a dollar sign:

```
PRINT USING "***$####,##";123.45
***$123.45
```

As before, the asterisks and the dollar sign in the format string count as place holders. You can use this method of formatting numeric output when you start writing checks with your Model 100!

You probably won't be surprised to learn that we can combine all these formatting techniques into a single PRINT USING statement. The following example shows how the federal government might use the Model 100 to write out a typical dollar amount on a check.

```
10 INPUT "amount needed"; DOLLARS
20 CHECKF$ = "***$#####,,##"
30 PRINT
40 PRINT USING CHECKF$; DOLLARS
RUN
amount needed? 87234762

****$87,234,762.00
OK
```

In some applications such as this one, we may want to direct the output to the printer rather than to the screen. Easy. Instead of using PRINT and PRINT USING statements, we use LPRINT and LPRINT USING. PRINT USING and LPRINT USING produce exactly the same output except that the former writes to the screen and the latter writes to the printer.

Specifying Exponential Notation

In scientific applications that deal with very large or very small numbers, we may wish to specify the format of numeric output in scientific or *exponential notation*. The PRINT USING statement can also do this job. For example, let's print out the value of the speed of light in various ways that use the powers of 10 discussed earlier.

```

10 C = 299792459 'meters/second
20 PRINT USING "##,##^ ^ ^ ^"; C
30 PRINT USING "##,####^ ^ ^ ^"; C
40 PRINT USING "###,###^ ^ ^ ^"; C
50 PRINT USING "####,##^ ^ ^ ^"; C
RUN
 3.00E+8
2.9979E+8
29.979E+7
299.79E+6
OK

```

The first two numbers in the output are in standard scientific notation; The only difference between them is in the number of significant digits shown, which is determined by the number of place holders after the decimal point in the format string. Note that the first number, which shows only three significant digits, has been rounded off. Two place holders are required in front of the decimal point, one for a digit, and the other for the sign of the number (as usual, a positive sign is not stated explicitly). Four caret symbols (^) are used as place holders for the power of 10 part of the number, such as E + 8 in our example.

The last two numbers in the output are in *exponential notation*, which, as you can see, is very similar to scientific notation; the only difference is that in scientific notation the first, or decimal, part of the number must be between one and ten, whereas in the more general exponential notation, this first part can have *any* size.

Summary of PRINT USING

The PRINT USING statement has the following syntax:

```
40 PRINT USING format string; N1, N2, N3, ...
```

This statement prints out the values of the variables listed after the semicolon (N1, N2, N3, ...) in a format specified by the *format string*. Table 9-2 summarizes most of the symbols that can be used within the format string to specify the format. Here is an example:

```

PRINT USING "###$#####,,##"; 1234.5678
###$1,234.57
OK

```

PRINT USING is an extremely versatile statement. We've covered most of the possible variations, but there are several we haven't discussed. With this introduction, however, you'll be able to use your manual to help solve any formatting problems we haven't covered.

Summary

In this chapter we focused on numbers. We explained that BASIC on the Model 100 distinguishes three types of numbers: double- and single-precision numbers and integers. Double-precision numbers have a maximum of fourteen digits, whereas single-precision numbers have a maximum of six digits. Both of these floating-point number types are frequently expressed in scientific notation and have a range of approximately 10^{62} to 10^{-64} . Numbers of the integer type are limited to the range of 32,767 to -32,768.

The three numeric types can be specified by means of either a type declaration tag (#, !, or % for double, single, or integer precision) or a type declaration statement (such as DEFINT) at the beginning of a program. The default numeric type on the Model 100 is double-precision, so double-precision numbers need not be type declared. In the last part of this chapter, we dealt with formatting numeric output by means of the PRINT USING statement. Different formats can be specified by using various symbols within a format string.

Symbol	Placement Within Format String	Function
#	Anywhere	Place-holder for number to be printed
.	Anywhere	Specifies position of decimal point
,	To left of decimal point	Causes commas to be printed every third digit to left of decimal point
\$\$	Left side	Appends \$ to number as prefix
**	Left side	Fills space to left of number with row of asterisks
^^^^	Right side	Causes number to be printed out in exponential notation

Table 9-2. A selection of formatting symbols that can be used within the *format string* of PRINT USING

Exercises

1. Write a short program that calculates your monthly salary given an input of the yearly salary. Use memory efficient variable types and print the result by means of the PRINT USING statement.
2. Rewrite the program "F->C" (which prints out a conversion table from degrees Fahrenheit to degrees Centigrade) from the "Solutions" section of Chapter 8, using the most memory-efficient variable types and the PRINT USING statement to print out the temperatures to an accuracy of one tenth of a degree.

Solutions

1. The following is one way to write this program:

```
10 INPUT "enter salary"; SAL!  
20 M,SAL! = SAL!/12%  
30 PRINT USING "$$#####,,##"; M,SAL!  
40 END  
OK
```

2. This is our modified version of "F->C":

```
10 REM--NAME;"F->C"-----  
20 REM  
30 PRINT "  DEG, F  DEG, C"  
40 PRINT "  -----"  
50 REM  
60 FOR F% = 60 TO 90  
70   C! = (F% - 32%)*5%/9%  
80   PRINT USING "   ##,##"; F%, C!  
90 NEXT  
100 END  
OK
```

RUN

Des. F	Des. C
--------	--------

60.0	15.6
------	------

61.0	16.1
------	------

62.0	16.7
------	------

63.0	17.2
------	------

64.0	17.8
------	------

*

*

*

OK

10

Character Graphics

Concepts

- The extended keyboard character set
- ASCII
- Constructing graphics images
- Moving characters
- Making boxes

Instructions

GRPH and **CODE** keys, CHR\$, MOD

One of the most interesting and enjoyable activities on the Model 100 is to write and experiment with programs that produce *graphics* output. The term *graphics* refers to any kind of visually interesting and meaningful display of information, such as a picture, a game-related image, or a graph. There are two different ways to produce graphic images on the Model 100. One is called *character graphics*, in which images are constructed out of ready-made characters available on the Model 100. Character graphics is the main topic of this chapter. The second method is called *dot graphics* or *all-points-addressable graphics*, in which individual dots on the screen can be turned off and on. Dot graphics will be the topic of a later chapter.

In the process of exploring character graphics, we'll also introduce the extended character set on the Model 100 and explore the ASCII character codes.

Of Pixels and Characters

Before we talk about graphics, we need to look at characters in general. A character is any symbol that appears on the screen when you press a keyboard key. If you look closely, you'll see that your screen is divided up

into an array of small squares, which are called *pixels* (or, more simply, dots), and that every character on the screen is constructed from these pixels. Figure 10-1 shows how the letters *R* and *s* are put together using pixels as building blocks. Each character, whether lowercase or uppercase, fits into a character space of six by eight pixels. Every character, even the period, is constructed from an array of six by eight pixels.

In character graphics, whole characters are assembled into a screen image. Although the usual keyboard characters (mostly letters and numbers) are sufficient for most text applications, the Model 100 has an additional repertoire of 130 *special symbols* designed to add variety and flexibility to the screen images. This “extended” character set includes mathematical symbols, foreign language letters, and symbols specifically designed for constructing graphics images. We will be using this third group of characters for most of our examples, though we will also provide an overview of the complete extended character set.

Direct Entry of Special Characters

There are two ways to generate special characters. The first is by a combination of the keys `GRPH` and `CODE`; this is the simpler method and we’ll describe it first. The second method, and the topic of the second part of this chapter, makes use of the ASCII codes to represent these characters.

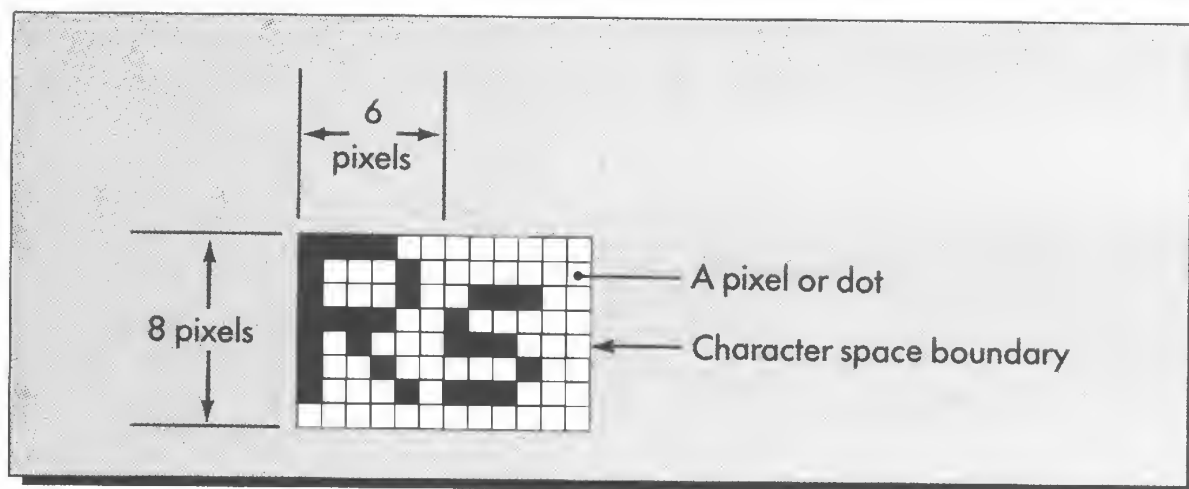


Figure 10-1. Characters and pixels

Graphics with **GRPH**

As an introduction to the extended character set, try the following. Locate the key marked **GRPH**, which stands for “GRaPH”. When you press this key by itself, nothing happens. However, if you press this key and hold it down while typing another key, say, the letter *q*, your Model 100 will print not a *q* but a little human figure! Try some other keys in combination with **GRPH** to get a sense of the great variety of characters you can generate in this manner. Other examples include a telephone (**GRPH** p), a house (**GRPH** h), a square root sign (**GRPH** r), and a heart (**GRPH** 4).

Also try *uppercase* characters in combination with **GRPH**; you’ll have to hold down both the **GRPH** and **SHIFT** keys. Many of these uppercase characters result in graphics characters that are particularly useful in printing images on your screen. For example, **GRPH** X (note that the letter X is a capital letter) prints a block the size of a character space. Table 10-1 shows a small sample of the characters available with the **GRPH** key. For a listing of all characters available on your keyboard, see the *TRS-80® Model 100 Portable Computer* manual or Appendix B.

The function of **GRPH** is to change the character that each keyboard key produces on the screen. In combination with **SHIFT** and **GRPH**, each










Keyboard Character	Printed Character
GRPH p	 (telephone)
GRPH h	 (house)
GRPH r	 (square root symbol)
GRPH q	 (person)
GRPH w	 (person)
GRPH 4	 (heart)
GRPH X	 (filled character space)
GRPH A	
GRPH Y	

Table 10-1. A selection of the extended character set with **CODE**

character key can print four different characters. The effects of some special keys, however, such as **TAB**, **SHIFT**, and **CTRL**, remain unaltered.

Using **GRPH** Within a BASIC Program

So far we've printed special characters by simply pressing a character key in combination with **GRPH**, but we can also include these special characters in a BASIC program. For example, the following program prints out a little person on the screen:

```
10 PRINT "⌘"  
RUN  
⌘  
OK
```



To enter character, press q while holding down **GRPH** key

BASIC “remembers” the graphics symbol and prints it out as if you had entered it yourself.

You can use the **PRINT @** statement to place a special character at a given location on your screen. Remember from Chapter 3 that screen locations are specified by the *screen coordinate*, a number between 0 and 319. For example, the command

```
PRINT @ 140, "⌘"
```

places the figure character near the center of your screen.

Sometimes it is convenient to specify the position of a character by means of a *row* and *column* position instead of the screen coordinate required by the **PRINT @** statement. The following program shows how this can be done:

```
10 REM---NAME:"ROWCOL,BA"-----  
20 REM---Prgm Prints house character  
30 REM---at specified ROW & COL  
40 CLS  
50 INPUT "row, column"; ROW%, COL%  
60 SCD% = (COL%-1) + (ROW%-1)*40  
70 PRINT @ SCD%, "⌘";  
80 END
```

This program assumes that the upper left corner of the screen is specified by ROW = 1 and COL = 1, so column positions can range from 1 to 39 and row positions from 1 to 8. However, you cannot print a character in the lower right corner of the screen because a carriage return will always cause the screen to scroll upward.

On the Model 100, BASIC's ability to print graphics characters that you can type into your program with the **GRPH** key makes possible a large variety of interesting graphics programs. The following are some simple examples that provide building blocks for more complex programs, especially game-related programs.

Building a House

Let's use what we've learned to build a house for our figure character (**GRPH** q):

```

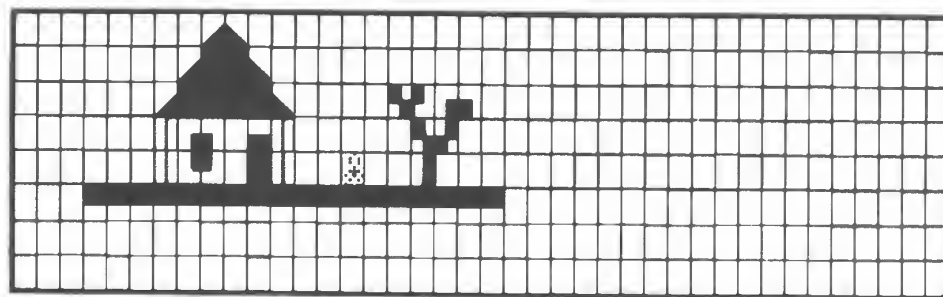
100 REM---NAME:"HOUSE,BAZ-----
120 REM---prgm draws a house, person,
130 REM---and tree
140 CLS
150 PRINT "
160 PRINT "
170 PRINT "
180 PRINT "
190 PRINT "
200 PRINT "
210 END

```



To construct the roof, we used the filled character space **GRPH** X and the filled triangles **GRPH** G and **GRPH** Y. Though the window looks like another filled character space box (**GRPH** X), it actually consists of two half boxes (**GRPH** Q is the upper half and **GRPH** W the lower half) that straddle two character spaces. The tree is made up of the characters **GRPH** %, **GRPH** ^, **GRPH** #, and **GRPH** W. You might enjoy adding a car, a mountain, and some clouds.

When run, this program produces a screen image very much like what is shown in the program:



A Boxed Title

We can also use graphics characters to help display information on the screen in an interesting way. For example, instead of simply printing out a centered title as we've done before, we can now use a box to give the title more emphasis, as in the following example:






```
10 REM---title with box-----
20 CLS
30 PRINT TAB(16) "  "
40 PRINT TAB(16) "  TITLE  "
50 PRINT TAB(16) "  "
60 END
```

Figure 10-2 identifies the keys used in combination with  that generate the box around "TITLE".

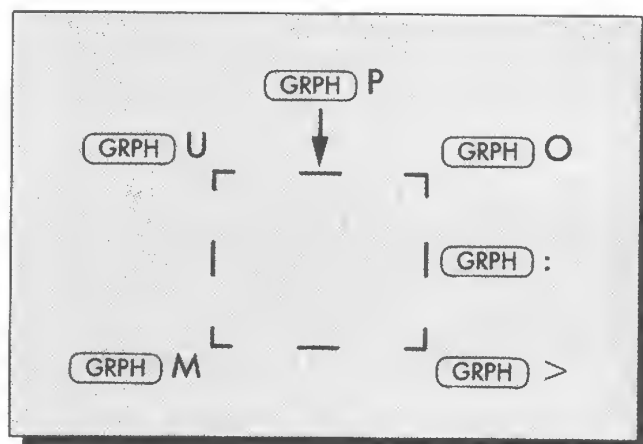


Figure 10-2.  codes used to construct a box

Moving a Character

The screen image in our previous example is a stationary one. It's not hard, however, to move a character on the screen. Consider the following program:

```
100 REM---NAME:MOV1-----
120 REM---Prgm causes figure to "run"
130 REM---across screen
140 REM
150 CLS
160 FOR J = 1 TO 280
170   PRINT @J,"&"
180   PRINT @J-1," "
190   FOR L = 1 TO 50: NEXT
200 NEXT
210 END
```

We can't show you the output of this program — it shows a little figure running (well, moving) across the screen — but this is how the program works. With each execution of the FOR...NEXT loop (lines 160-200), the figure character printed by line 170 advances to the screen coordinate defined by the new value of J. The following statement in line 180 prints a space (" ") at position J-1; its effect is to erase the figure character in a screen position behind the one where line 170 just printed the most recent figure character. The result is the appearance of a figure moving across the screen! The FOR...NEXT loop in line 190 slows down the program and, consequently, the running figure. Without it, the figure zooms across the screen very rapidly, making it difficult to see.

Extended Characters with **CODE**

The characters keys in combination with the **GRPH** key provide us with a large variety of special characters, many of which are especially well suited for graphics. However, we can print an additional forty-four characters by means of the **CODE** key. **CODE** is used in exactly the same manner as **GRPH**: you hold it down while pressing a character key. Try a few examples. You'll find that most of them print letters with umlauts and accent marks, letters that for the most part belong to German and French. As with the characters generated by **GRPH**, characters printed with **CODE** can

be used within a program. For example, we can enter the German word for *suspenders* in response to INPUT and then print it as follows:

```
10 INPUT "a foreign word"; WD$
20 PRINT WD$
RUN
A foreign word? Hosenträger
Hosenträger
OK
```

Press the a key while holding down **CODE** key

This program “remembers” the letter *a* with the umlaut and prints it out just as if we had typed it with the key combination *a* and **CODE**.

Using the **GRPH** and **CODE** Keys

By using the **GRPH** and **CODE** keys in combination with other character keys, we can extend the standard keyboard character set to include a great variety of special symbols useful in mathematics, foreign languages, and graphics. The following example illustrates how we can use the **GRPH** key to generate a small figure:

$\left\{ \begin{array}{l} \text{Pressing q while holding} \\ \text{down the } \mathbf{GRPH} \text{ key} \end{array} \right\}$ causes the π symbol to appear

We can use the **CODE** key in the same manner, and we can use symbols generated in combination with the **GRPH** or **CODE** key within programs, as shown in the following example:

```
10 PRINT "π"
RUN
π
OK
```

Press p while holding down **GRPH** key

The Extended Character Set with ASCII

You’ve now seen how to generate any of the special characters available on the Model 100 by using character keys in combination with the **GRPH** or **CODE** key. However, you can also print out any of these characters with

the BASIC instruction CHR\$ and a special code for all the characters on your Model 100, called *Extended ASCII*.

ASCII stands for “American Standard Code of Information Interchange”. Basically, it’s a code that associates a number with each character. All 220 printable characters, as well as 35 nonprintable characters, can be identified by an ASCII code number. Appendix B lists the ASCII code numbers and associated characters, called ASCII values. From Appendix B, you can see that the letter *a* has the ASCII code 97 and the figure character that we previously printed using (GRPH) q has the code number 147. Table 10-2 shows a selection of characters and their ASCII code numbers.

The following are a few general remarks about the way code values are assigned to characters. Code numbers 0-32 don’t return any printed characters; they are reserved for special communication instructions between computers and computer peripherals. For example, the ASCII code number 13 means “do a carriage return”.

Codes 33-126 are identified with all the usual keyboard characters, such as letters and numbers. Codes 0-126 have the same meaning for practically all computers and communications equipment and constitute the original ASCII standard. Codes 127 to 255 are Tandy’s own embellishment of ASCII. These are the code values associated with all the characters we can print by using the (GRPH) and (CODE) keys. The whole character set from code numbers 0 to 255 is called *Extended ASCII*.


The CHR\$ Function — Putting ASCII to Good Use

Now that we have a numeric code for each printable character, how can we use this code to print characters? What we need is the BASIC instruction

ASCII Code Number	ASCII Character	Comments
1	Zilch, nothing!	Codes 0-32 don’t return printable characters
35	#	
53	5	Digits 0-9 have codes 48-57
65	A	Capital letters: codes 65-90
97	a	Lowercase letters: codes 97-122
147	⌘ (figure)	Same as (GRPH) q
239	■ (solid ch.sp)	Codes 225-255 are specially designed for graphics displays

Table 10-2. Examples of the ASCII character set

CHR\$, which translates an ASCII code number into the corresponding ASCII character. This type of instruction is called a *string function*, and though we'll devote a later chapter to this topic, we'll introduce it now so that we can use ASCII to generate graphics output.

The following example shows how to use CHR\$ to print out the little figure we generated earlier using  q.

```
10 PRINT CHR$(147)
RUN
*
OK
```

The number 147 is the ASCII code number for the figure character. The function CHR\$ translates this code number into the corresponding character which is then printed to the screen by PRINT. Any printable character can be printed out in this manner.

Instead of printing the value of CHR\$(147) directly with the PRINT statement, as we have done above, we can first assign the value of CHR\$(147) — the figure character — to a string variable and then print the value of the string variable, as in this example:

```
10 FIG$ = CHR$(147)
20 PRINT FIG$
RUN
*
OK
```

Note that CHR\$ returns a *string* value, so the variable to which it is assigned must also be a string value.

The CHR\$ Function

The CHR\$ function converts an ASCII code number to a corresponding character, called the ASCII value. For example, the statement

```
10 PRINT CHR$(128) ← 128 is the ASCII code number for the
                      telephone symbol
```

returns a character that looks like a telephone. For printable characters, ASCII code numbers can range from 33 to 255. CHR\$ must be part of a PRINT-related statement, or it must be assigned to a string variable.

Displaying the Whole ASCII Character Set

Because we can refer to each ASCII character by a number, we can use the CHR\$ function as part of a FOR...NEXT loop to display the whole extended ASCII character set:

```
10 REM---NAME:ASCII.BA-----
20 REM---Prgm displays extended ASCII
30 REM---character set
40 CLS
50 FOR J = 33 TO 255
60   PRINT CHR$(J) + " ";
70 NEXT
80 END
```

Try this program on your Model 100 to see what all these characters really look like. Note that we added a string containing a space (" ") to the value of CHR\$ in line 60: its purpose is to leave some "elbowroom" between the characters so that we can tell where one leaves off and another begins.

Programming a Jumping Figure

As the previous program, ASCII.BA, demonstrates, using the CHR\$ function to display ASCII characters makes it easy to change characters while a program is being executed. One example that makes good use of this concept is the following program, which displays a little figure on the screen that appears to jump up and down:

```
100 REM---NAME:JUMP.BA-----
120 REM---Prgm displays figure that
130 REM---appears to jump up and down
140 REM
150 CLS
160 PRINT @ 131, " ██████████ "      ← Line keyed in with GRPH Q
170 FOR J = 1 TO 100
180   PRINT @95, CHR$(147 + J MOD 2)
190   FOR L = 1 TO 60: NEXT L
200 NEXT J
210 END
```

Again, you should try this program yourself to see what it does. This is how it works. The PRINT @ statement in line 160 prints out a "ground" line on which the figure can jump. The figure itself is printed by the PRINT @ statement in line 180. Note that the screen coordinate (the number after @) is the fixed value 95, which is located right above the ground line printed by line 160. The secret to creating the effect of a figure jumping is contained

inside the parentheses of the CHR\$ function: as each loop is executed, the value of $(147 + J \text{ MOD } 2)$ alternates between 147 and 148. ($J \text{ MOD } 2$ returns the remainder or “MODulus” of the integer quotient $J/2$ — that is, 0 if J is even or 1 if J is odd.) The ASCII character for code 147 is the little figure we’ve used before; the ASCII character for code 148 is *another* figure, which has his or her feet up in the air. The jumping effect, then, is created by alternating the two figures.

These examples provide some building blocks with which to write your own graphics displays. The possibilities for creating interesting effects are virtually endless.

Summary

In this chapter we have introduced the extended character set, which includes various special characters in addition to the usual keyboard characters. Our primary interest has been in characters that are especially useful as building blocks for graphics images, which can in turn be used to construct screen displays for games, text, and graphs.

We can print special characters in two ways: (1) we can simply press a character key with or without **SHIFT**, while simultaneously holding down either the **GRPH** or **CODE** key; or (2) we can use the CHR\$ function to translate an ASCII code number into a character value, which we can then print with PRINT or PRINT @.



Exercises

1. Write a program that prints a pulsing heart character at the center of the screen.
2. Write a program that moves a character figure diagonally across the screen.
3. Write a program that asks the user to input numbers from 0 to 8 and then makes a bar graph of the entered values. The program should accept twelve values.

Solutions

1. To create the heart, we’ll use CHR\$ and the ASCII code number 158. (Note that we could instead use **GRPH** 4.)

```

10 REM---Pulsing Heart-----
20 CLS
30 FOR J = 1 TO 100
40   PRINT @140, CHR$(158)      ← Prints heart
50   FOR T = 1 TO 80: NEXT      ← Pause
60   PRINT @140, " "            ← Erase heart
70   FOR T = 1 TO 80: NEXT      ← Pause
80 NEXT
90 END

```

2. The character we'll move diagonally across the screen looks like a racing car.

```

100 REM---NAME:"MVDIAG,BA"-----
110 REM---Racing car moves diagonally
120 REM---across the screen
130 CLS
140 FOR J = 0 TO 6
150   PRINT @J*44, CHR$(132);    ← Prints car
160   FOR T = 1 TO 60: NEXT T    ← Pause
170   PRINT @J*44, " ";         ← Erases car
180 NEXT
190 PRINT @80, " ";             ← Locates cursor on third line
200 END

```

3. Using the filled character with ASCII code number 239, we've come up with the following program:

```

100 REM---NAME:"BARGRH,BA"-----
110 REM---Prgm draws 12 bars; values
120 REM---between 0 & 8 determined by
130 REM---user input
140 CLS
150 COL = 10
160 FOR BR = 1 TO 12
170   COL = COL + 2              ← Advances to new bar position
180   PRINT @0, "               " ; ← Erases previous input information
190   PRINT @0, " ";           ← Moves cursor to upper left corner
200   INPUT "Value"; VA         ← Requests user input
210   IF VA = 0 THEN 270        ← If input = 0, then don't draw bar
220   TP = 9 - VA              ← Top row position of bar
230   FOR ROW=8 TO TP STEP -1   ← Draws one bar
240     PS=(COL-1)+(ROW-1)*40
250     PRINT @PS, CHR$(239);   ← 239 is ASCII code for GRPH X
260   NEXT ROW
270 NEXT
280 PRINT @ 40, " "            ← Places cursor on second line
290 END

```

Subroutines and Program Organization

Concepts

- Subroutines
- Structured programming
- Indexed subroutines
- Interrupt subroutines

Instructions

- GOSUB, RETURN, ON...GOSUB, KEY() ON, ON...KEY GOSUB, KEY() OFF, KEY() STOP

All the programs you've written so far have been quite short — less than a page long. As your programs grow longer, however, it becomes increasingly important to maintain a logical order and structure. Writing a long and complex program is in some ways like building a house: both may be rather overwhelming projects unless broken down into smaller, more manageable tasks. In this chapter we explore one of the principal methods for defining such subunits of a program — the subroutine.

The Subroutine — The Concept

A *subroutine* is a program within a program; it is a set of instructions that is performed whenever it is *called* by the main program. A subroutine might, for example, PRINT a title page, move a character on the screen, or graph the data calculated by the main program. In our house analogy, a

subroutine is like a subcontract for the windows: they are ordered, built, and delivered ready-made when "called" for.

Aside from helping divide your program into smaller, more manageable units, subroutines can be very useful when your program needs to perform a series of instructions several times. Suppose you're writing a game program that requires placing five dragons on different parts of your screen. Instead of writing the part of the program that draws the dragon five times, you can write a single subroutine that draws the dragon. Then you can use this subroutine to draw as many dragons as needed.

If you feel that we're trying to impress you with the importance and utility of subroutines, you're right! In addition to making programming easier, sophisticated statements such as ON...KEY GOSUB allow a user to interact with the program to create such effects as firing a laser beam or moving a race car character around on the screen by pressing certain keys. Before we program such effects, however, let's take care of the basics.

GOSUB...RETURN and the Basic Subroutine

The GOSUB statement is the most common way to call a subroutine. Like the GOTO statement, it results in branching to a specified line number. However, one important difference makes the GOSUB statement particularly useful in calling subroutines and keeping order within your programs. See if you spot the difference in the following program:

```
10 CLS
20 PRINT "Part 1 of main program"
25 '=====
30 GOSUB 100
35 '=====
40 PRINT "Part 2 of main program"
45 '=====
50 GOSUB 100
55 '=====
60 PRINT "Back to the main program--bye"
70 END
80 '
100 PRINT "  This beep is brought to"
110 PRINT "  you by a SUBROUTINE!"
120 BEEP
130 RETURN
```

```

RUN
Part 1 of main Program
  This beep is brought to you
  by a SUBROUTINE!                ← Model 100 beeps
Part 2 of main Program
  This beep is brought to you
  by a SUBROUTINE!                ← Model 100 beeps
Back to the main Program--bye
OK

```

The output shows how this program works. GOSUB 100 in line 30 tells your computer to “GO to the SUBroutine that starts at line 100”; so execution of your program branches to line 100, which causes your computer to print the message “This beep is brought to you by a SUBROUTINE!” and to beep. So far, that’s exactly the same thing that the statement GOTO 100 would have done.

The next statement to be executed is the RETURN in line 130, which causes program execution to return to the original branch point. Line 40 now prints “Part 2 of main program”. RETURN always causes program execution to branch back to the line *immediately following* the most recent GOSUB statement.

Line 50 then branches to line 100, the beginning of the subroutine. After the subroutine is executed for the second time, program execution returns to line 60 of the main program, which prints the message “Back to the main program — bye”. In general, a subroutine can be “called” by the main program as many times as you wish. As you can see, subroutines are very useful whenever you wish to execute the same sequence of statements at various places in a program.

The statements GOSUB and RETURN are usually used in *combination*. Though it is theoretically permissible to have a GOSUB without a RETURN statement, it is the combination of the statements that makes them particularly well suited to define and call a subroutine: GOSUB 100 causes branching to line 100, the *beginning* of the subroutine; RETURN defines the *end* of the subroutine and returns program execution to the original branch point.

Now the difference between GOTO and GOSUB becomes evident: GOSUB, like GOTO, causes branching; unlike GOTO, however, it also *remembers* where to branch *back to* when RETURN is executed. Most programmers prefer to use GOSUB statements in place of GOTO statements wherever possible, in order to avoid the messy convolutions in program flow that become all too likely when too many GOTOs are used.

Returning to our previous example, we can illustrate the flow of our program as shown in Figure 11-1.

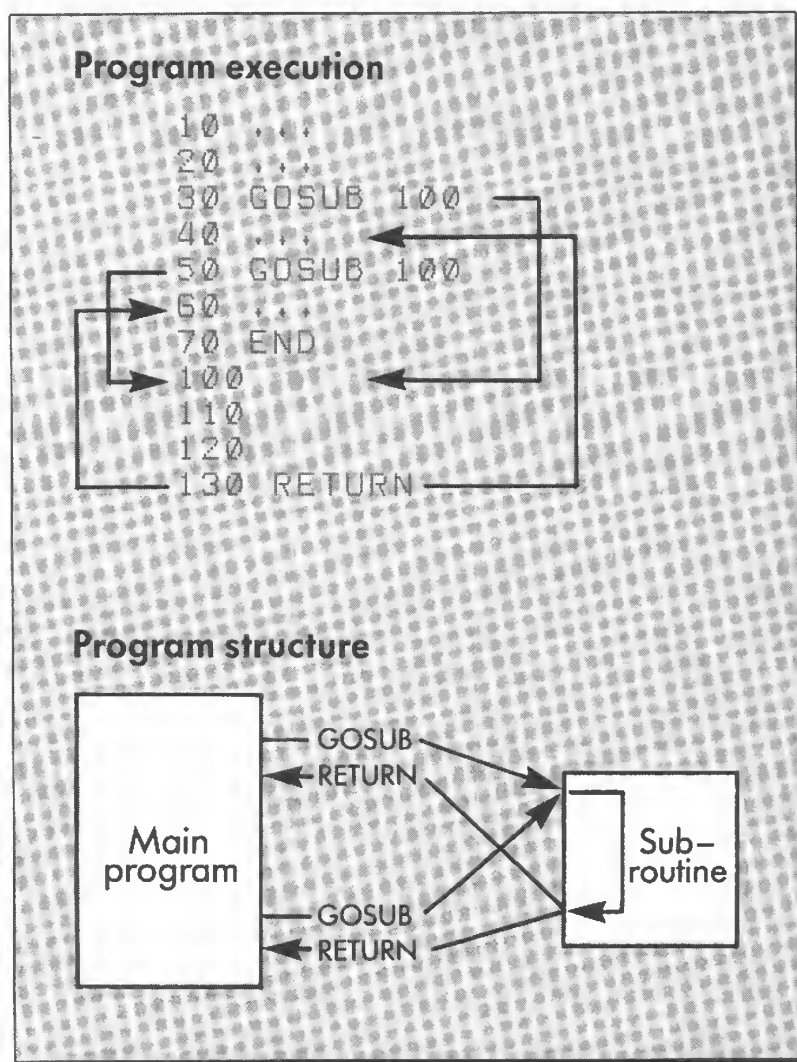


Figure 11-1. Program flow of a subroutine

The following summary generalizes what we've learned from this example.

The GOSUB and RETURN Statements

The statement

```
1000 GOSUB 1000
```

causes program execution to branch to line 1000, which is the beginning of a subroutine. A GOSUB statement must be accompanied by a RETURN statement of the form

```
1500 RETURN
```

which causes program execution to "return" to the statement immediately following the GOSUB statement. RETURN defines the end of a subroutine.

An Example — Organizing Your Program Using GOSUB and RETURN

Using a subroutine to print a message and cause the computer to beep illustrates the use of GOSUB and RETURN statements, but it does little to illustrate why subroutines are useful. After all, we could have written this program without the subroutine. Though subroutines rarely make sense in short programs, they are very helpful in organizing longer programs and in breaking them down into smaller, more manageable units.

To demonstrate the usefulness of subroutines, let's write a program that draws a castle, a frightened maiden, and an advancing dragon.

```
100 REM--NAME:SUB1.BA-----
110 REM--prgrm draws castle with a
120 REM--dragon, & a frightened maiden
130 REM
140 CLS
150 '=====
160 GOSUB 1000                ← Draws castle
170 '=====
180 '
190 FOR SC%=185 TO 172 STEP -1 ← Draws maiden and advancing dragon
195 '=====
```

```

200  GOSUB 2000                                ← Draws jumping maiden
210  '=====
220  GOSUB 3000                                ← Draws dragon at screen coordinate SC%
230  '=====
240  NEXT
250  PRINT @100, "gulp";
900  END
901  '
1000 REM==SUBR draws castle=====
1005 '
1010 PRINT
1020 PRINT "      "
1030 PRINT "      "
1040 PRINT "      "
1050 PRINT "      "
1060 PRINT "      "
1100 RETURN
1110 '
2000 REM==SUBR draws frightened maiden=
2005 '
2010 FOR J% = 1 TO 10
2020   PRINT @170, CHR$(147 + J% MOD 2);
2030   FOR P% = 1 TO 160: NEXT
2040 NEXT
2100 RETURN
2110 '
3000 REM==SUBR draws dragon=====
3005 '
3010 PRINT @SC%-80, "      "
3020 PRINT @SC%-40, "      "
3030 PRINT @SC%   , "      "
3100 RETURN

```

Notice how this program is organized. The main program consists mainly of three GOSUB statements that call subroutines to draw a castle, a maiden, and a dragon. (Figure 11-2 identifies the key combinations used to construct the castle and dragon.) The first subroutine, which draws a stationary castle, is called only once. The second and third subroutines, however, which draw the maiden (she is jumping up and down) and the advancing dragon, are called with *each* execution of the FOR...NEXT loop. The index of this loop is the screen coordinate of the lower left corner of the dragon figure. (See Chapter 10 to review screen coordinates.) Each subroutine is a logically distinct unit that performs a very specific task. The main program serves as a “manager”, and the subroutines do the actual job of drawing the screen images.

Using subroutines to divide this program into self-contained units makes it easy to understand, because you needn't grasp every detail (how the maiden is made to appear frightened, for example) in order to get a general sense of what the program does. Once you get an overview of the program by reading the main program, you can always dissect the individual subroutines to understand the details.

Indexed Branching with ON...GOSUB

The ON...GOSUB statement is similar to the GOSUB statement in that both call subroutines. However, whereas GOSUB calls only the subroutine specified by the line number following GOSUB, ON...GOSUB chooses any one of a number of specified subroutines according to an index value. ON...GOSUB is useful whenever a program needs to branch to one of several different subroutines.

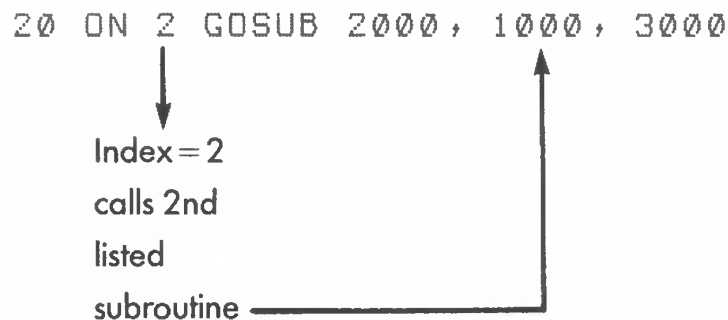
The following program demonstrates how the ON...GOSUB statement works:

```
10 INPUT INDEX
20 ON INDEX GOSUB 2000, 1000, 3000
30 END
40 '
1000 PRINT "Subroutine 1000 called by INDEX = 2"
1010 BEEP
1020 RETURN
1030 '
2000 PRINT "Subroutine 2000 called by INDEX = 1"
2010 BEEP
2020 RETURN
2030 '
3000 PRINT "Subroutine 3000 called by INDEX = 3"
3010 BEEP
3020 RETURN
RUN
? 2
Subroutine 1000 called by index = 2      ← Computer also beeps
OK
```

In this particular RUN, our response to the INPUT question mark assigned the value 2 to the variable INDEX. The output of this program demonstrates that the called subroutine begins with line 1000. In addition, it suggests that this particular subroutine (rather than one of the other two) is "chosen" by the value of the variable INDEX, in this case 2. Note that this subroutine is the second one listed right after GOSUB. Aha! So that's how

ON...GOSUB works: the INDEX value 2 causes the program to select the *second* subroutine listed after GOSUB. If INDEX has the value 1, the program should branch to the first subroutine listed, which begins with line 2000; similarly, if INDEX equals 3, the program should branch to the *third* subroutine listed. Note that the line numbers listed after GOSUB don't have to be in an ascending order.

The following diagram illustrates how ON...GOSUB made its selection in the previous example:



Let's run the above program a few more times to explore some of the fine points of ON...GOSUB:

```

RUN
? 1.8
Subroutine 2000 called by INDEX = 1
OK
RUN
? 0
OK
  
```

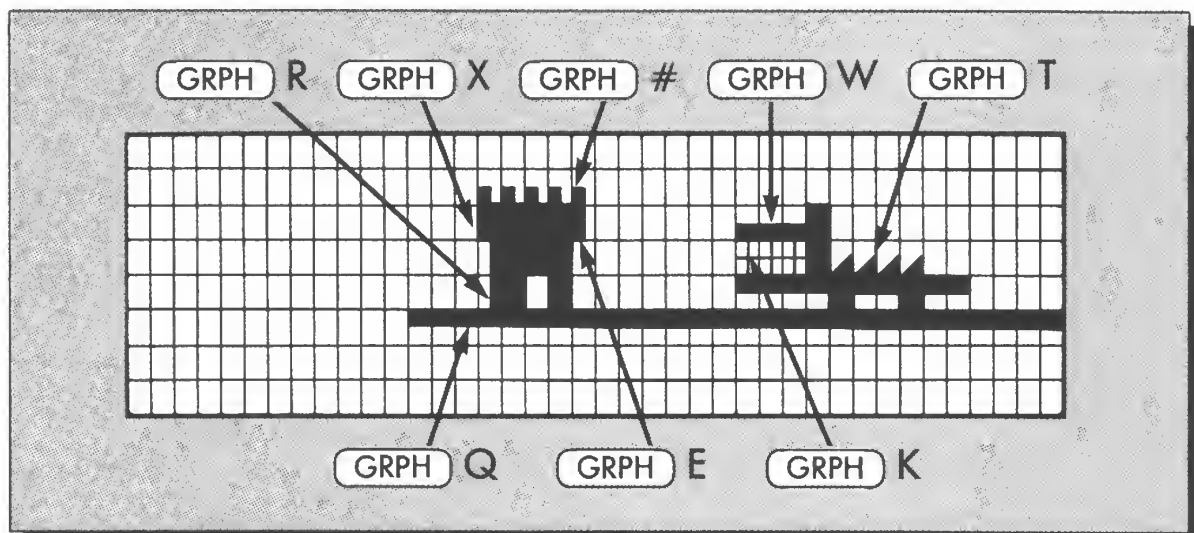


Figure 11-2. Characters used to construct the castle and dragon in the program SUB1.BA

The first RUN demonstrates that ON...GOSUB *truncates* (drops the digits after the decimal point) the value of INDEX to make its choice of subroutines: the value we entered for INDEX was 1.8, but the value used to choose the subroutine was 1. Similarly, had we entered 2.1, the second subroutine (line 1000) would have been chosen.

The second RUN shows what happens when there is no match between the index and the listed subroutines: the index we entered is 0, and there is no subroutine listing for zero (there's a first, second, and third but none for zero). The result is that the program ignores the ON...GOSUB statement and proceeds to the next line, the END statement. The same thing happens when an index equal to or larger than 4 is entered. Thus an index equal to zero or greater than the number of line numbers listed after ON...GOSUB causes program execution to "fall through" to the next line.

Writing a Menu Program — An Example Using ON...GOSUB

When you go to a restaurant to eat dinner, you look at the menu, make a choice, and place your order with the waiter. You certainly don't want to go into the kitchen, wash the lettuce, scrub the potatoes, and clean the escargots. You go to a restaurant precisely because you *don't* want to get involved in the details of cooking and in the mess of cleaning up afterward.

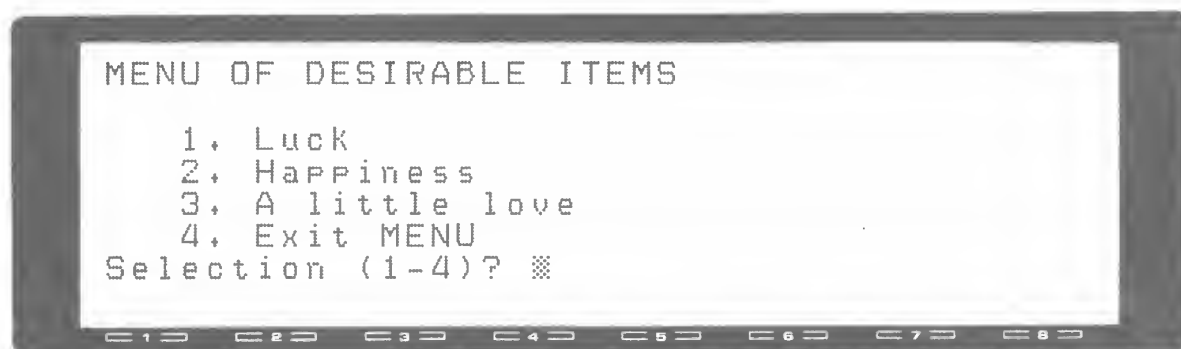
Many people feel pretty much the same way about using computers: the typical user probably doesn't want to get involved in the details of operating and programming a computer just to run a program; this user wants to see what programs are available and run them without having to learn such commands as LIST, LOAD, and RUN. A *menu* of programs, from which he or she could choose as easily as we tell a waiter what he want, would be of great benefit to this typical user.

To illustrate how such a menu might be written, we'll write a program that (1) displays a menu that includes a title, a list of selections (including an "exit menu" option), and an explanation of how to make the selection; (2) branches to the selected subroutine, using an ON...GOSUB statement; (3) has each subroutine display something on the screen for a short time; and (4) has program execution RETURN to the menu. A serious set of programs concerning such topics as finances, growth models, or learning math would most likely be very long. We've chosen a fairly short example (the main program is only 20 lines long) that uses some of the graphics techniques you learned in Chapter 10. And, well, it's not terribly serious.

Here's what our program looks like:

```
100 REM--NAME:MENU1,BA-----
110 REM--Prgrm requests user input to
120 REM--select one of 3 subroutines
130 REM--listed in a menu & exit option
150 CLS
152 REM--menu page-----
154 REM
160 PRINT "MENU OF DESIRABLE ITEMS"
170 PRINT
180 PRINT "    1. Luck"
190 PRINT "    2. Happiness"
200 PRINT "    3. A little love"
210 PRINT "    4. Exit MENU"
230 INPUT "Selection (1-4)"; SEL
232 REM
234 REM--subroutine selection-----
236 REM
240 ON SEL GOSUB 1000,2000,3000,900
250 GOTO 150
900 END '-----Program end-----
910 '
1000 REM==subr. 1 (Luck)=====
1010 CLS
1020 FOR P = 80 TO 240
1030   PRINT @P, CHR$(156) 'clover lf,
1040 NEXT
1100 RETURN
2000 REM==subr. 2 (Happiness)=====
2010 CLS
2020 FOR J = 1 TO 20
2030   PRINT @140, CHR$(147+J MOD 2)
2040   FOR T = 1 TO 50: NEXT
2050 NEXT
2100 RETURN
3000 REM==subr. 3 (A little love)=====
3010 CLS
3020 FOR J = 1 TO 20
3030   PRINT @140, CHR$(158) 'heart
3040   FOR T = 1 TO 60: NEXT
3050   PRINT @140, " "
3060   FOR T = 1 TO 60: NEXT
3070 NEXT
3100 RETURN
```

When run, this program presents the following menu:



The above menu is straightforward and unambiguous — if a bit questionable in content! We did assume that the user knows what **ENTER** means. For a general use program, however, even that should be explained.

How does this program work? Don't be overwhelmed by its length. The main program is very simple and short; most of the program is taken up by subroutines that are called by the main program.

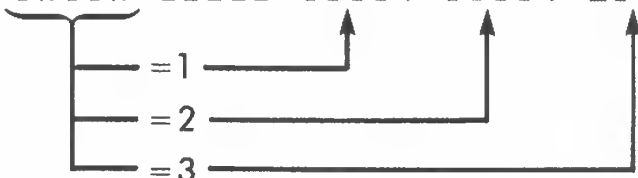
The main program (which ends at line 900) does the following. Lines 152-230 print the menu. Line 230 asks for user input to the variable SEL, which is then used as the index in the ON...GOSUB statement in line 240 to control branching to one of the listed subroutines. For example, if SEL is assigned the value 2 (by user response to INPUT), then the second listed subroutine (beginning with line 2000) is chosen. If the user enters the number 4, then the fourth listed subroutine is called (the END statement in line 900). Notice that this particular "subroutine" doesn't end with the customary RETURN. Generally, all subroutines should end with a RETURN: you won't get an error message if you don't include one, but you run the risk of confusing and messing up the internal workings of BASIC. (One exception to this rule is the special case in which the subroutine executes an END statement.)

The three main subroutines (1000, 2000, and 3000) consist of graphics statements with which you're already familiar, so we won't go into any details about them. Even though you can easily figure out what they do, go ahead and RUN this program — you'll enjoy it!

The ON...GOSUB Statement

The ON...GOSUB statement selects a subroutine according to the value of an index variable in a manner illustrated by the following statement:

```
20 ON index GOSUB 1000, 5000, 2000
```



That is,

An index value = 1 calls the *first* listed subroutine (line 1000)

An index value = 2 calls the *second* listed subroutine (line 5000)

An index value = 3 calls the *third* listed subroutine (line 2000)

And so forth

If the index has a truncated value greater than the number of line numbers listed or if its value is less than 1, then program execution falls through to the next line.

Calling a Subroutine Anytime with KEY() ON and ON...KEY GOSUB

In the last section you saw how an index value determines which of several subroutines will be selected by an ON GOSUB...RETURN statement. We can also select a subroutine at any time during program execution by pressing any one of a number of specially “activated” function keys. The BASIC statements that perform this function are very similar to ON...GOSUB...RETURN, though an additional statement — the KEY() ON statement — is required to activate a particular key (or set of keys) so that it will function in the desired manner. The KEY() ON and GOSUB...RETURN statements are ideal for applications that require the user to interact with the computer during program execution.

The following example shows how this pair of statements works:

```
10 KEY(1) ON
15 KEY(2) ON
20 ON KEY GOSUB 1000, 2000
30 GOTO 30
40 '
1000 PRINT "Subroutine #1 called"
1020 RETURN
1030 '
2000 PRINT "Subroutine #2 called"
2020 RETURN
RUN
Subroutine #2 called      ← We pressed key (F2)
Subroutine #1 called      ← We pressed key (F1)
Break in 30               ← We pressed (CTRL) (BREAK)
```

In the above example, we first pressed the function key (F2), and the program immediately responded with “Subroutine #2 called”. When we pressed the function key (F1), the program again responded immediately, this time with “Subroutine #1 called”. We could continue to press either (F1) or (F2) as often as we wished, and each time one of these messages — depending on which key we pressed — would be printed out. To terminate program execution, we pressed (CTRL) (BREAK).

Let’s see how this program works. The first two statements are KEY(1) ON and KEY(2) ON. These tell the Model 100 to “turn on” or to “activate” function keys (F1) and (F2) (rectangular keys at the upper left of your keyboard). Activating keys (F1) and (F2) causes the following ON...GOSUB statement in line 20 to respond to keys (F1) and (F2) while the program is running. If we don’t activate keys (F1) and (F2) in this way, pressing these keys will have no effect during program execution. That is, the subsequent ON...KEY GOSUB 1000, 2000 statement simply won’t “work”, in the same way that pushing a car’s gas peddle has no effect unless the engine is turned on with the ignition key.

The ON...KEY GOSUB statement in line 20 works in a way similar to the familiar ON *index* GOSUB statement: in the ON *index* GOSUB statement, the value of the *index variable* determines which of the listed subroutines will be called, whereas in the ON...KEY GOSUB statement, the *function key number* determines which of the listed subroutines will be called. For example, pressing (F1) causes the first listed subroutine to be called, (F2), the second, and so forth.

What's the purpose of the endless loop in line 30? If it wasn't there, program execution would reach the end of the program long before we could press one of the function keys. The reason is that the ON...KEY GOSUB statement doesn't wait for the user to press a key; execution proceeds right on to the next BASIC line. However, the Model 100 remembers that it had encountered an ON...KEY GOSUB statement and will execute it *whenever* an appropriate function key is pressed. In contrast to most other BASIC statements, which are active only at the time of their execution, once it has been read by the computer, ON..KEY GOSUB can interrupt the normal program flow at any point within the main program (but not within a subroutine). This process of calling a subroutine at any time by pressing a particular key is sometimes called *key-trapping*.

Now the purpose of the endless loop in line 30 becomes clear: because the Model 100 reads the first three lines of the program very quickly, the endless loop simply keeps the computer busy while the user decides which function keys to press. When we press a function key, (F1) or (F2) in our example, program execution temporarily steps out of this loop to do the job you requested, which is to execute subroutine 1000 or 2000. After that job is done, the program RETURNS to its looping, "waiting" for the next time function key (F1) is pressed.

Note that we've been talking about *pressing* function keys, not *entering* function keys. After both KEY() ON and ON...KEY GOSUB have been read by the computer, the listed subroutine will be called immediately when the activated key is pressed.

Which keys can be activated by a KEY() ON statement? The answer is that we can activate (F1), or any sequence of function keys beginning with (F1), up to (F8). But we cannot leave any "holes" in the sequence; that is, we can't activate (F1), (F2), and (F5) without also activating (F3) and (F4). If we want to activate all eight function keys at once, we use the single statement

```
10 KEY ON
```

in place of eight separate KEY() ON statements. Note that none of the usual keyboard keys can be "activated", only the eight function keys.

The KEY() ON and ON...KEY GOSUB Statements

The combination of KEY() ON and ON...KEY GOSUB statements causes branching to one of several listed subroutines whenever an activated function key is pressed. For example, the statements

```
100 KEY(1) ON          ← Activates key (F1)
110 KEY(2) ON          ← Activates key (F2)
120 KEY(3) ON          ← Activates key (F3)
    *
    *
    *
250 ON KEY GOSUB 1000,2000,3000
```

cause subroutine 1000, 2000, or 3000 to be called *whenever* the function key (F1), (F2), or (F3) is pressed, no matter what instruction the program is executing. The first three KEY() ON statements activate the function keys (F1), (F2), and (F3) so that the ON...KEY GOSUB statement will respond when one of these keys is pressed.

KEY() OFF and KEY() STOP — Deactivating the Keys

Whenever a program ends or is terminated with (CTRL) (BREAK) (as was the case in our example), all the previously activated keys are automatically restored to their functions in the BASIC Command Mode. For example, (F1) will once again list RAM files (unless, of course, we've changed the function of this key, as described in Chapter 4).

Occasionally, we need to deactivate previously activated keys while the program is still running. One way to do this is with the KEY() OFF statement, where the number in parentheses indicates which key is being deactivated. For example,

```
120 KEY(1) OFF
```

has the effect of deactivating key (F1).

We can also deactivate a key with the KEY() STOP statement, which differs from KEY() OFF in that it “remembers” if the key has been pressed; when the key is reactivated with KEY() ON, the subroutine associated with the “remembered” key is then called. For example,

```
120 KEY(1) STOP
```

will deactivate key **(F1)**. If we press key **(F1)**, no program interruption and subroutine call takes place. However, if the statement

```
200 KEY(1) ON
```

appears later in the program, key **(F1)** will be reactivated, and the subroutine associated with key **(F1)** will be called immediately. We use KEY() STOP whenever our program needs to *temporarily* shield itself from key interruptions.

An Example Using ON...KEY GOSUB

The KEY() ON and ON GOSUB statements allow a program to be interrupted and to respond immediately to user input. Games are one application in which this responsiveness is especially valuable. The following program illustrates how the Model 100 can enable a user to move a laser about the screen and, on command, fire a laser beam. This program is not in itself a complete game, but a building block for generating your own “Space-Wars”.

The following is our program “LASER1.BA”:

```
100 REM--NAME: "LASER1.BA"-----
110 REM--Prgrm allows user to move
120 REM--a laser up and down via [F1] and
130 REM--[F2] keys, and fire laser via
140 REM--[F3] key,
150 REM
160 DEFINT P,J,T
200 CLS
205 REM--initialize variables-----
206 REM
210 P = 82 'initial coord of laser
220 BEAM$ = CHR$(241) 'beam character
230 LAS$ = CHR$(239)+CHR$(241) 'laser
240 PRINT @P, LAS$ 'laser at begin,
```

```

250 REM
260 REM--select subroutines-----
270 REM
280 KEY(1) ON          'activate [F1]
290 KEY(2) ON          'activate [F2]
300 KEY(3) ON          'activate [F3]
310 ON KEY GOSUB 1000,2000,3000
320 GOTO 320           'endless loop
330 END
340 '
1000 '==SUBR, moves laser up=====
1005 '
1010 PRINT @P," " 'removes old laser
1020 P=P-40        'new laser coord,
1030 PRINT @P,LAS$ 'new laser
1100 RETURN
1110 '
2000 '==SUBR, moves laser down=====
2005 '
2010 PRINT @P," " 'removes old laser
2020 P=P+40        'new laser coord,
2030 PRINT @P,LAS$ 'new laser
2100 RETURN
2110 '
3000 REM==SUBR, fires laser beam=====
3005 '
3010 PL=P+2        'initial beam coord
3020 FOR J=0 TO 32 'advances beam
3040   PRINT @PL,BEAM$ 'prints beam
3050   FOR T=1 TO 10:NEXT
3060   PRINT @PL," " 'erases beam
3070   PL=PL+1      'advances beam coord,
3080 NEXT J
3090 RETURN

```

This program is fairly long, but its organization makes it easy to understand. The first part assigns the variables BEAM\$ and LAS\$ to define characters that make up, respectively, the laser beam (a short horizontal line) and the laser itself (a filled character space plus a horizontal line segment). The second part of the program handles the key-trapping that enables keys **(F1)**, **(F2)**, and **(F3)** to call subroutines 1000, 2000, and 3000. The first two subroutines move the laser character LAS\$ up and down by printing spaces (" ") over the old laser position, updating the laser coordinate P, and printing the new laser character LAS\$ at the new coordinate P.

The third subroutine animates the laser beam moving across the screen. Variable PL defines the starting screen coordinate of the beam character BEAM\$; the FOR...NEXT loop causes the beam character to advance to the right by printing BEAM\$ at coordinate PL, pausing for a moment, then erasing the old BEAM\$ character, and finally advancing the beam coordinate PL. (Don't move the laser off the screen; the program doesn't check for this.)

If you have any inclination to write a game program, this example will undoubtedly give you many ideas of games or game-related screen images. If your interest lies in another area, say, finances or science, this example can provide a model that you can adapt to your own programming needs.

Summary

We've been primarily concerned in this chapter with the subroutine and how it can be called by the main program. A subroutine is a subprogram that handles a specific task, such as printing a screen image, moving a character across the screen, or handling a specific mathematical calculation. The simplest way to call a subroutine is with the GOSUB and RETURN statements: GOSUB tells the program which line to branch to, and RETURN tells the program to "return" to the statement immediately following the GOSUB statement.

The ON...GOSUB statement calls one of several listed subroutines; the particular subroutine chosen depends on the value of the index variable listed after ON. This statement is particularly useful in menu programs, in which a user entry determines which of several subroutines is to be executed.

The combination of KEY() ON and ON...KEY GOSUB statements enables a program to immediately call one of several listed subroutines when one of the function keys has been pressed. A function key must be "activated" by the KEY() ON statement; then the ON...KEY GOSUB statement selects a subroutine *whenever* an activated function key is pressed, even though the program is executing another statement at the time.

Exercises

1. Rewrite the bar graph program BARGRH.BA (given as a solution to exercise 3 in Chapter 10) and use two subroutines. One subroutine should

print out a title page with the title and an explanation of what the program does; the second subroutine should draw each bar.

2. Write a program that allows a user to move a character, say, a race car, around the screen in any of the four basic directions. (This program may be useful as a subroutine in your own programs.)

Solutions

1. The following is our bar graph solution using subroutines. This program is listed here as it would be printed out on a line printer with LLIST.

```
100 REM--NAME:"BARGH2,BA"-----
110 REM--Prgrm draws 12 bars with
120 REM--height determined by user
130 REM--input, Uses subroutines
135 DEFINT C, B, P, R
140 '=====
150 GOSUB 1000 'title
160 '=====
170 CLS
180 COL=10
190 FOR BAR = 1 TO 12
200   COL=COL+2
210   PRINT @0,"          "
220   PRINT @0,"";
230   INPUT "value"; VA
240   IF VA=0 THEN 280
250   '=====
260   GOSUB 2000 'draws bar
270   '=====
280 NEXT
290 PRINT @40, ""
300 END
310 '
1000 '==SUBR. to draw title page=====
1010 CLS
1020 PRINT TAB(15) "BAR-GRAPH"
1030 PRINT
1040 PRINT "This program graphs 12 values as bars"
1050 PRINT "between the values of 0 to 8"
1060 FOR P=1 TO 2000: NEXT
1100 RETURN
```

```

1110 /
2000 '==SUBR, draws bar=====
2005 /
2010 TP = 9 - VA
2020 FOR ROW = 8 TO TP STEP -1
2030   P = (COL-1) + (ROW-1)*40
2040   PRINT @P, CHR$(239);
2050 NEXT
2100 RETURN

```

You may wish to add another subroutine that draws a vertical scale with markings from 0 to 8.

2. The following program allows the user to move a race car character (ASCII code 133) around on the screen, using keys **F1** through **F4**.

```

100 REM--NAME:"MOVE1,BA"-----
110 REM--Prgrm allows user to move car
120 REM--via Keys [F1] through [F4]
130 REM
140 CLS
150 C$ = CHR$(133) 'race-car
154 P = 140 'start, coord
156 PRINT @P, C$
160 KEY(1) ON: KEY(2) ON
170 KEY(3) ON: KEY(4) ON
180 ON KEY GOSUB 1000,2000,3000,4000
190 GOTO 190
200 END
210 /
1000 '==SUBR, left=====
1005 /
1010 PRINT @P, " "
1020 P=P-1
1030 PRINT @P, C$
1040 RETURN
1050 /
2000 '==SUBR, right=====
2005 /
2010 PRINT @P, " "
2020 P=P+1
2030 PRINT @P, C$
2040 RETURN
2050 /
3000 '==SUBR, up=====
3005 /
3010 PRINT @P, " "
3020 P=P-40
3030 PRINT @P, C$

```

```

3040 RETURN
3050 '
4000 '==SUBR, down=====
4005 '
4010 PRINT @P, " "
4020 P=P+40
4030 PRINT @P, C$
4040 RETURN

```

The variable `P` defines the screen coordinate. The variable `C$` is ASCII code 133, which is the race car character. Each subroutine first erases the character at `P`, then updates `P`, and finally prints a race car character at the new coordinate `P`. Note that the screen coordinate `P` must be increased by 40 to move the car down one line and decreased by 40 to move the car up one line.

12

Data and Arrays — Organizing Information

Concepts

Efficient storage of data

Arrays

Subscripts

Dimensions

Instructions

DATA, READ, RESTORE, DIM, ERASE

*I*n chapter 5 you learned about variables — how to use them and how to assign values to them. A variable is a symbolic representation of one unit of information, such as a number or a phrase (string). Many programming problems, however, require the handling of large bodies of information: values must be assigned to many variables, and large blocks of these variables must be manipulated in some way. In this chapter we discuss the efficient storage of data by means of DATA and READ statements, as well as the powerful method of organizing and manipulating information using *arrays*.

Storing and Reading Information with DATA and READ

Suppose we are writing a program that requires us to assign the number of a month to a variable having the name of that month. We can fulfill our objective by writing twelve statements, of which the first few look like this:

```

10 JAN = 1
20 FEB = 2
30 MAR = 3
  *
  *
  *

```

That's fine, although these statements take up twelve lines in our program. But what if we needed to assign calories to one thousand different food variables? We'd need to write one thousand program lines! There must be a better way to do this, and there is. BASIC comes to the rescue with the DATA and READ statements.

The following program makes the same month assignments as our previous statements, but it uses DATA and READ:

```

10 DATA 1,2,3,4,5,6
20 READ JAN, FEB, MAR, APR, MAY, JUN
30 PRINT JAN; FEB; MAR; APR; MAY; JUN
40 END
RUN
 1  2  3  4  5  6
OK

```

From this output, you can see that JAN has the value 1, FEB the value 2, and so forth. It's easy to see how these assignments were made: the numbers following DATA in line 10 are values that are assigned in their proper order to the variables listed after READ in line 20. For example, the third variable listed after READ (MAR) is assigned the third value listed after DATA (3). We can visualize this assignment process like this:

```

10 DATA  1,      2,      3,      4,      5,      6
          ↓        ↓        ↓        ↓        ↓        ↓ (Arrows show
          JAN      FEB      MAR      APR      MAY      JUN assignment)
20 READ JAN, FEB, MAR, APR, MAY, JUN

```

Notice that the DATA statement doesn't really "do" anything; it's a *nonexecutable statement*. The computer simply stores the information listed after DATA. The READ statement actually "READs" the list of DATA values and assigns them in order to the variables listed after READ.

The example above is straightforward, but there are many different ways to use DATA and READ, with a corresponding number of do's and don'ts.

Matching the DATA and READ Lists

What if we don't READ the whole list of values after DATA? That's just like buying only some of the items on your grocery list; if you decide you want all the items, you can always go back for the rest. Try it with the previous program: eliminate the last variable in the READ statement (JUN) and run the program. The output will be the same as before, except that the last number printed out will be 0, the value of JUN. Because no value was explicitly assigned to JUN, BASIC (as usual) assigns the default value 0.

What if we run out of DATA values — which happens if the number of variables listed after READ exceeds the number of values listed after DATA? Let's try it:

```
10 DATA 1, 2, 3, 4, 5          ← We eliminated the sixth data value
20 READ JAN, FEB, MAR, APR, MAY, JUN
30 PRINT JAN; FEB; MAR; APR; MAY; JUN
40 END
RUN
?OD Error in 20
OK
```

The letters *OD* in the error message “?OD Error in 20” stand for “Out of Data”. The message is clear: we ran out of DATA because the sixth variable (JUN) listed after READ has no corresponding sixth value in DATA.

Additionally, when using DATA and READ, the *variable type* must match the *value type*. Because reading DATA is basically a process of assigning values to variables, all the rules for assigning variables apply, including the requirement of type matching. Consider the following example, in which READ assigns both numeric and string variables:

```
10 DATA grapes, tripe, "apple pie", 55
20 READ FRUIT$, YUCK$, DESSERT$, NUM
30 PRINT FRUIT$, YUCK$, DESSERT$, NUM
40 END
RUN
grapes      tripe
apple pie   55
OK
```

There's nothing surprising here except that we used quotation marks around “apple pie” — or rather, that we didn't use quotation marks to identify the other string values. Previously, when we assigned a string constant to a string variable by means of an equal (=) sign, we *always* had to bracket the string constant with quotation marks. This requirement is relaxed in the

DATA statement: if the variables listed after READ are string variables, READ “knows” that it should interpret the values listed after DATA as string constants. For one-word string constants, such as the first two foods in our example, quotation marks are optional. However, quotation marks are required for the last string constant listed — “apple pie” — because it contains a space. Generally, quotation marks around a string constant are optional *unless* the string constant contains spaces or punctuation that BASIC would otherwise confuse with its own vocabulary of characters, such as commas and semicolons.

Multiple DATA Statements

Because DATA is a nonexecutable statement, we can place it anywhere in our program. When READ is executed, it will look for and find the DATA statement, wherever it is. We probably needn’t point out, however, that in a well-organized program we wouldn’t put our DATA just *anywhere*. The beginning of a program, near the end, or adjacent to the READ statements are some preferred locations. Also, we can use more than one DATA statement. The important point here is that READ will read a *series* of DATA statements as *one list*. To illustrate this idea, let’s break the DATA statement in our example program into two separate statements:

```
10 DATA 1, 2, 3
20 READ JAN, FEB, MAR, APR, MAY, JUN
30 PRINT JAN; FEB; MAR; APR; MAY; JUN
40 DATA 4, 5, 6
50 END
RUN
 1  2  3  4  5  6
OK
```

Works just fine! In addition to splitting our original DATA statement into two DATA statements (lines 10 and 40), we placed the second DATA statement *after* the READ statement to illustrate that it doesn’t matter where DATA is located. READ always interprets the values listed after DATA as a *single list*, even though this list may be split into several different DATA statements in different locations. It is sometimes useful to interpret the way the READ statement reads DATA in terms of an imaginary arrow that points to the data item to be read. Each time after a value is read, the pointer moves to the next data item, regardless of where it is in the program. Figure 12-1 illustrates the way DATA is read in our previous program.

Multiple READ Statements

Several READ statements may “read” a DATA list. Let’s split up the READ statement in our program:

```
10 DATA 1, 2, 3, 4, 5, 6
20 READ JAN, FEB, MAR
30 READ APR, MAY, JUN
40 PRINT JAN; FEB; MAR; APR; MAY; JUN
50 END
RUN
 1  2  3  4  5  6
OK
```

No problem: after the first READ statement (in line 20) is finished reading JAN, FEB, and MAR, the second READ statement (in line 30) takes over and reads APR, MAY, and JUN. The second READ statement gets in line after the first one and keeps the pointer moving as if all the variables were listed after one READ statement.

The DATA and READ Statements

DATA and READ can appear in a program in the following way:

```
100 DATA 1, 2, 3
110 DATA 4, 5, 6
   *
   *
   *
230 READ A, B, C, D
```

DATA statements are nonexecutable statements that form a single list of data that can be read by one or more READ statements.

READ assigns values from the DATA list to the corresponding variables listed after READ. In the example above, variables A, B, C, and D are assigned the values 1, 2, 3, and 4, in that order. Multiple READ statements have the same effect as a single READ statement that has the combined list of variables.

Number of Days Elapsed Since January 1 — An Example

If you want to calculate the amount of interest earned since you invested your money, one thing you’ll need to know is the number of days that your

money has been invested. Many business (and scientific) calculations require that you know the number of elapsed days between two given days. We won't give a general solution to this problem, but we will provide a building block — a program that finds the number of days that have elapsed between January 1 and any date in the same year. We'll use a DATA statement to list the number of days in each month:

```

10 REM--NAME:ELAPDY.BA-----
20 REM--Prgm finds days elapsed since
25 REM--Jan, 1 to entered date
30 REM
100 DEFINT A-Z
105 DATA 31,28,31,30,31,30,31,31,30      ← Number of days in each month
110 DATA 31,30,31
115 INPUT "Month (1-12)-----"; MNTH
117 INPUT "Day of month (1-31)"; DOM
120 DAYS = 0
130 FOR M = 1 TO (MNTH - 1)              ← Finds days to first of month
140     READ DM
150     DAYS = DAYS + DM
160 NEXT M
165 DAYS = DAYS + DOM - 1                ← Finds total number of days
170 PRINT DAYS "days have elapsed since Jan, 1"
180 END
RUN
Month (1-12)-----? 3
Day of month (1-31)? 15
 74 days have elapsed since Jan, 1
OK

```

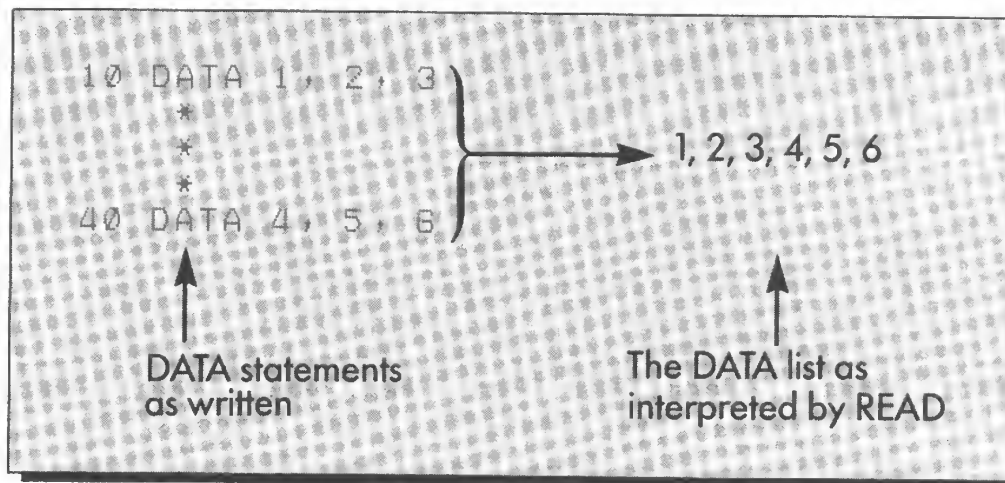


Figure 12-1. Successive DATA statements form a single list

Here's how the program works. The DATA statements list the number of days in each month. When the user enters the number 3 in response to INPUT in line 115, the value 3 is assigned to the variable MNTH (we can't use MONTH because it contains the Reserved Word ON). Because (MNTH - 1) is the index limit of the FOR...NEXT loop, the loop will now be executed two times — for values of M equal to 1 and 2. The first time the loop is executed, the READ statement (line 140) reads the first number in DATA — the number of days in January — and the following statement (line 150) adds that number to the existing value of DAYS, which is zero. So the new value of DAYS is 31, the number of days in January. The second time the loop is executed, line 140 reads the second number in DATA — the number of days in February — and line 150 adds that value to the existing value of DAYS. The value of DAYS is now 31 + 28, which equals 59 — the number of days up to March 1. In general, every time the loop is executed, it causes the number of days in the next month to be added to the existing total of DAYS. Finally, we add the number of days in the current month, 15 in this example.

This program provides a good example of how we can use DATA and READ statements to store and read a fixed body of information.

RESTORE — Going Back to the Beginning

Sometimes we need to READ a DATA list more than once. For example, if our previous program were part of a subroutine that was called more than once, the READ statement would try to read the DATA list as many times as the subroutine was called. But there's a hitch: remember that every time *any* READ statement reads a value from a DATA list, it reads the value after the last value that was read (the pointer just keeps on going down the list); therefore, when READ is executed a second time, all the data have already been read, and the computer will return a “?OD Error in xx” message.

The way out of this difficulty is to use the RESTORE statement before you reread a DATA list. This resets the data pointer to the start of the list.

The following example shows how to READ the variables JAN, FEB, and MAR three times:

```
10 DATA 1, 2, 3, 4, 5, 6
20 FOR J = 1 TO 3
30   READ JAN, FEB, MAR
40   PRINT JAN; FEB; MAR
50   RESTORE
60 NEXT
70 END
RUN
  1  2  3
  1  2  3
  1  2  3
OK
```

← Restores pointer to beginning of DATA list

That's the output we want: no matter how many times READ assigns values to JAN, FEB and MAR, we always want those values to be 1, 2, and 3. The RESTORE statement “restores” the data pointer to the first value in the DATA list so that the second (and third) time the READ statement is executed (in the second and third loops), it again begins reading the DATA list from the beginning: JAN again is assigned the first value in the DATA list, FEB, the second, and so forth. It might be helpful to see what would happen if we neglected to include the RESTORE statement. The following is the output of our previous program with the RESTORE statement in line 50 deleted:

```
RUN
  1  2  3
  4  5  6
?OD Error in 30
OK
```

That makes sense, right? The data are read sequentially until we run out of data.

Arrays, Subscripts, and Dimensions

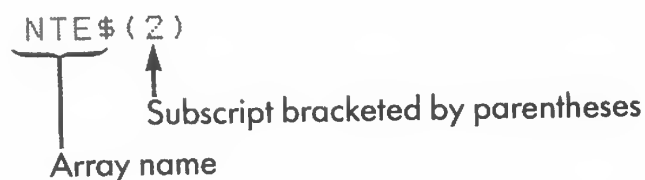
Different variables represent different quantities. So far, we've done something that seems perfectly obvious: we've used different variable names to represent these different variables (or quantities). For example, the statements

```
10 DATA DO, RE, MI
20 READ N1$, N2$, N3$
```

assign values DO, RE, MI to the variables N1\$, N2\$, N3\$, which clearly have different names. We chose very similar variable names, however, to indicate that all three variables represent similar quantities — musical notes. In applications like this, where we need to assign and manipulate related groups of variables, it is often convenient and sometimes essential to use a similar but much more powerful approach using *arrays*.

Arrays — An Introduction

An *array* is a group of variables, all of which have the same “array” name. The different variables (or members) within the group are called *array elements*, and are identified by a number called the *subscript*. An example of an array element is



where NTE\$ is the array name (*NTE* stands for “NOTE”, which we cannot use because it contains the Reserved Word *NOT*) and the 2 in parentheses is the subscript. The subscript in an array is *always* bracketed by parentheses. Other array elements have the *same name* but *different subscripts*. For example, the next element in the array NTE\$ is NTE\$(3).

Arrays can be assigned values in the same way that “normal” variables are assigned values. For example, the values DO, RE, MI can be assigned to an array using the familiar DATA and READ statements:

```
10 DATA DO, RE, MI
20 READ NTE$(1),NTE$(2),NTE$(3)
30 PRINT NTE$(1),NTE$(2),NTE$(3)
RUN
DO          RE
MI
OK
```

The output verifies that elements of the array NTE\$, defined by subscripts 1, 2, and 3, have been assigned the values DO, RE, MI. Figure 12-2 shows that the array elements of NTE\$ can be thought of as a sequence of adjacent variable boxes filled with the values of the array elements; each array element is identified by the name of the array and the subscript.

Using DATA and READ statements as we've done in the above example, however, is usually not the best way to assign values to array elements. This is especially true when dealing with arrays that have a large number of elements. Because array elements can be referred to by their subscripts, a more sophisticated approach to assigning and reading an array is the use of a FOR...NEXT loop:

```
10 DATA DO, RE, MI, FA, SO, LA, TI, DO
20 FOR J% = 1 TO 8
30   READ NTE$(J%)
40 NEXT
50 '
60 FOR J% = 1 TO 8
70   PRINT NTE$(J%) + " ";
80 NEXT
90 END
RUN
DO RE MI FA SO LA TI DO
OK
```

Line 30 reads successive array elements with each execution of the FOR...NEXT loop in lines 20-40. That's possible only because each array element is identified by its subscript, which in this example equals the loop index J%. Similarly, the loop in lines 60-80 causes the PRINT statement in line 70 to print successive array elements.

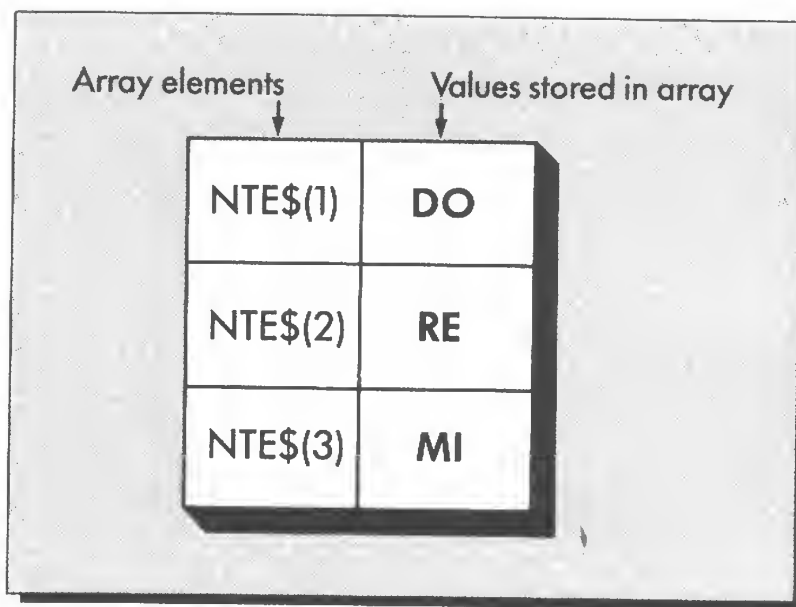


Figure 12-2. Variable boxes of the array NTE\$

Notice the convenience of using the FOR...NEXT loop to assign and print out an array. It certainly would have been tedious had we separately assigned each array element using only the DATA and READ statements, as we did in the earlier example. The advantage of the FOR...NEXT loop becomes even more obvious when we consider the prospect of assigning and printing out a really large number of values, say, 100!

Arrays are also ideal tools with which to manipulate or rearrange a sequence of numeric or string values. But before we offer more examples, we need to discuss a few more basics, such as rules for array names and subscripts, and the DIM statement.

Rules for Array Names and Subscripts

The rules for naming arrays are identical to those for naming variables. Our array NTE\$ ends with a dollar sign (\$) because the values we assign to it are string constants. Array names can also have the standard numeric type declaration tags — the % for integers, the ! sign for single-precision numbers, and the # for double-precision numbers (although the last is optional because double precision is the default value of the Model 100). Because arrays, especially large ones, require a significant amount of memory, it is often important to use the most memory efficient array type consistent with a particular application. Also, because an array name is the same for all array elements, all values assigned to an array must be of the same type — we can't mix integer and string values unless we have an appetite for error messages!

We also assign default values for array elements in the same way that we do for “normal” variables. This means that if we print STCKVL(5) without having assigned a value to it, we'll get a 0.

What are the rules for subscripts? Subscripts can be of any numeric type, although most applications use integers. Decimal subscripts are truncated to integer values; for example, the array element CAT(3.7) is identical to CAT(3). Subscripts can also be variables; for example, the element CAT(X) is identical to CAT(3) if the value of X equals 3.

The smallest subscript that we've used in our examples is 1, which agrees with our sense of the “first” array element or value. However, try the following BASIC command:

```
10 PRINT MNEY(0)
RUN
0
OK
```

The computer's response to the above command suggests that the zeroth element (defined by the subscript 0) does indeed exist; otherwise, we probably would have gotten an error message. The value of MNEY(0) is zero because we didn't assign a value to it. The Model 100 always assumes that zero is the lowest array element. However, just because it exists doesn't mean we have to use it; we can ignore this element, although we waste a little memory if we do.

What's the largest subscript value we can use? The answer has two parts: if we use arrays as we've done so far (without what is called a *dimension declaration*), then the largest subscript is 10; but if we do make a dimension declaration (as we'll explain in the next section), the size of the largest subscript (and, consequently, the number of elements in the array) is limited only by the amount of memory in the computer.

Defining Arrays with the DIM Statement

An array represents an ordered group of values that is stored in the computer in adjacent memory locations. In order to keep other values from "butting" into the array, the computer needs to reserve space in its memory for the whole array. Unless we indicate otherwise, the Model 100 will assume that any array we start using will have subscripts ranging from 0 to 10, and it will reserve eleven adjacent memory locations. On the other hand, we can explicitly request a specific amount of memory space (usually more than eleven locations) by means of the DIM statement.

The following example shows how this is done. The statement

```
10 DIM NTE$(12)
```

notifies the computer that we're going to use a string array called NTE\$ and that it should reserve a block of memory for thirteen array elements — that is, for elements 0 through 12. This DIM statement must be executed *before* the array is initialized or manipulated. A DIM statement for a particular array can be executed only *once* within a program. "Of course", you say, "why would I write it twice?" Well, chances are you wouldn't, but sometimes a program tries inadvertently to *execute* it a second time via some kind of loop; then you'll get the error message

```
?DD Error in 10
```

which means "you've got Duplicate Definition in line 10".

Dimensions of Arrays

Aside from informing the computer of the name, type, and upper limit of the subscript, the DIM statement has another important function: it declares the *DIMension* of the array. The *dimension* of an array equals the number of subscripts that it has. The arrays we've discussed so far have just one subscript — hence all have been one-dimensional arrays. You can visualize a one-dimensional array as a list of elements (and corresponding values) that can be laid out in one dimension, that is, on a line.

The previous DIM NTE\$(12) statement defines the dimension of the array NTE\$ to be 1 because only one subscript is present (the number 12). However, the statement

```
10 DIM TTT$(3,3)
```

defines a two-dimensional array called TTT\$ (TTT stands for “Tick-Tack-Toe”), because it lists the upper value of *two* subscripts (3 for both). Each element of this array is identified by two subscripts, as, for example, in the element TTT\$(2,3). A two-dimensional array can be imagined as a surface or grid, in this case a three-by-three tick-tack-toe template. Figure 12-3 identifies all the elements of the array TTT\$. Each square of this tick-tack-toe template corresponds to an element of the array TTT\$; rows of this array are defined by the first subscript, columns by the second. For example, the element TTT\$(2,3) refers to row 2 and column 3. The *value* of an array element might be an “X” or a “0”. For example, the assignment

```
20 TTT$(3,2) = "X"
```

could represent an “X” mark on the square defined by row 3 and column 2.

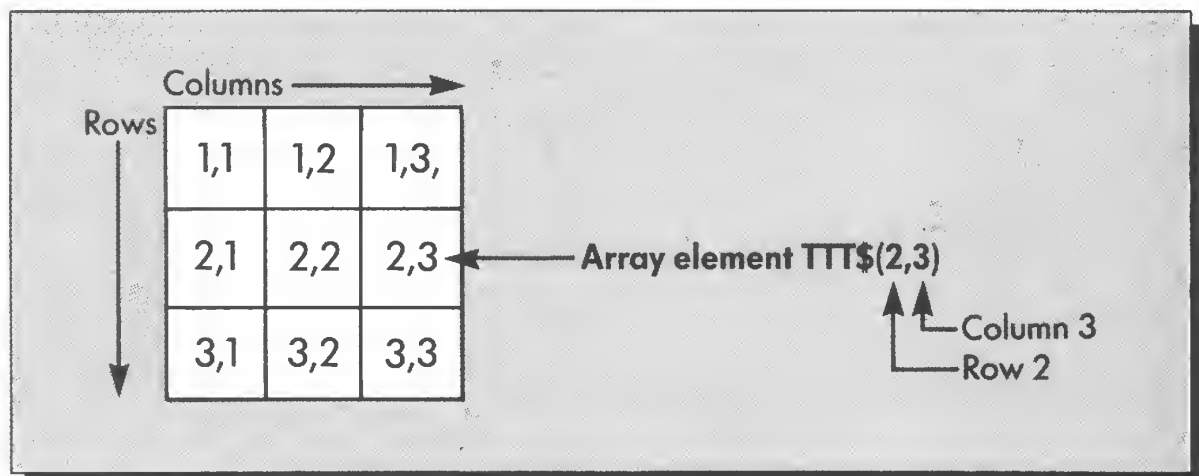


Figure 12-3. The two-dimensional array TTT\$

Table 12-1 lists some other examples of two-dimensional arrays.

It is possible and often quite useful to define arrays of more than two dimensions. For example, a three-dimensional array would be required to represent a particular configuration of a Rubicks cube. The location of each square on the cube could be represented by *three* subscripts, such as X, Y, and Z, and the *value* of each array element would be the name of the color: “red”, “white”, or “blue”.

Arrays and the DIM Statement

An array is a group of variables that have the *same* variable name; members of the array, called *elements*, are differentiated and identified by one or more subscripts. The *dimension* of an array equals the number of subscripts in its elements. For example,

INCME (NAM , MNTH)

The diagram shows the text 'INCME (NAM , MNTH)' with three curly braces underneath. The first brace is under 'INCME' and has a vertical line pointing down to the label 'Name of array'. The second brace is under 'NAM' and has a vertical line pointing down to the label 'Variable name of first subscript'. The third brace is under 'MNTH' and has a vertical line pointing down to the label 'Variable name of second subscript'.

is an element in the array INCME (for “INCoME”) having the subscripts equal to the values of NAM (for “NAME”) and MNTH (for “MoNTH”). Because two subscripts are present, this element belongs to a two-dimensional array.

Before using an array, it is a good practice (and imperative, if the largest subscript is to be larger than 10, or if the array is multi-dimensional) to define or dimension an array with the DIM statement, as shown here:

DIM INCME (30 , 12)

The diagram shows the text 'DIM INCME (30 , 12)'. A curly brace is under 'INCME' with a vertical line pointing down to the label 'Name of array to be dimensioned'. An upward-pointing arrow is under '30' with the label 'Largest value of first subscript'. Another upward-pointing arrow is under '12' with the label 'Largest value of second subscript'.

This DIM declaration reserves memory space for a two-dimensional array having 30 and 12 as the maximum values for its two subscripts.

Examples Using One-Dimensional Arrays

Before we leave the topic of arrays, we'd like to present two examples that illustrate their power and versatility. To keep our examples simple, we've used only one-dimensional arrays, although the concepts to be developed can easily be extended to multidimensional arrays.

That array elements can be referred to by their subscripts is the key to their usefulness in handling large volumes of related information. Consider the following program, which asks the user to enter some letters and then prints out the same letters in *reverse* order:

```
100 REM--NAME:"REVERS,BA-----
110 REM--prgm requests user input of
120 REM--number of letters and then the
130 REM--letters; prints letters in
140 REM--reverse order
150 REM
200 INPUT "Number of letters"; NUM
210 DIM L$(NUM)           ← Number of letters = largest subscript
220 FOR J = 1 TO NUM       ← Initializes array
230   INPUT "letter"; L$(J) ← Each letter assigned to one array element
240 NEXT
250 '
255 PRINT "Letters in reverse ";
257 PRINT "order are: ";
260 FOR J = NUM TO 1 STEP -1 ← Prints out array in reverse order
270   PRINT L$(J);
280 NEXT
290 END
```

Subject	What Array Represents	The Array
A seating chart	Name vs. row and column	NME\$(ROW,COL)
Stock values	Price of stock vs. company and month	STCK(CO,MO)
Rain fall	Inches rain vs. month and year	RAIN(MO,YEAR)
Grocery prices	Prices vs. item and market	PRICE(ITEM,MARKET)
A video picture	Color vs. row and column	COLR(ROW,COL)
Multiplication table	Product vs. row and column	N(ROW,COL)

Table 12-1. Examples of two-dimensional arrays

```

RUN
Number of letters? 4
letter? s
letter? t
letter? o
letter? p
Letters in reverse order are: pots
OK

```

Line 210 of this program dimensions the array L\$ so that the number of letters to be entered is equal to the largest subscript number. This procedure assures that enough memory will have been reserved so that each letter to be entered can be assigned to an array element. Lines 220-240 ask for user input of a letter and assign this letter to an array element. Lines 260-280 print out the array in reverse order by starting the FOR...NEXT loop with the largest loop index (NUM) and ending up at the smallest index (1).

The previous program illustrates two ways that arrays can be very useful. First, arrays are ideal vehicles for storing large bodies of flexible or nonpermanent information (in this example, letters entered by the user). Second, an array is perfect for storing a unit of information that must be manipulated in some ordered fashion (in our example, values of the array L\$ printed in reverse order). Array elements can also be manipulated in other ways. For example, array elements can be interchanged, a process useful in some sorting routines; in two-dimensional arrays, columns and rows can be added together or multiplied by a certain factor, a procedure applicable in spreadsheet-type programs.

Our last example is more complex than the previous examples. The problem is to write a program that reads a list of numbers and finds the smallest one. By itself, this program may seem unimportant, but it is an essential building block of many sorting routines — programs that rearrange numbers in an ordered sequence (starting with the smallest, for example) or that rearrange letters or words in alphabetical order. The following is a program that finds the smallest number in the sequence listed in DATA:

```

10 REM--NAME:SMALLST,BA-----
20 REM--Prgm finds lists sequence of
30 REM--10 numbers and finds smallest
40 '
100 DIM NUMBER(10)
110 DATA 5,6,4,9,14,3,7,14,9,10

```

```

115 PRINT "The 10 numbers are:"
120 '
130 FOR J = 1 TO 10
140     READ NUMBER(J)
150     PRINT NUMBER(J);
160 NEXT
170 '
180 TEST = 1000                ← TEST must be larger than any number in DATA list
190 FOR J = 1 TO 10
200     IF NUMBER(J) < TEST THEN TEST = NUMBER(J)
210 NEXT
220 PRINT
230 PRINT "The smallest number =" TEST
240 END
RUN
The 10 numbers are:
 5  6  4  9 14  3  7 14  9 10
The smallest number = 3
OK

```

Sure enough, 3 is the smallest number in this list. The first part of this program is old hat by now: the array `NUMBER` is defined and then initialized with the given `DATA`, and the values of its elements (the `DATA`) are printed. Line 180 begins the routine that tests for the smallest number in the array. It assigns the value 1,000 to the variable `TEST`, a “yardstick” to which the elements in the array will later be compared. The size of this number doesn’t really matter, as long as it is larger than any of the numbers in the array. Line 200 is the key statement in this program: it checks to see whether the array element `NUMBER(J)` is smaller than `TEST`. If it is not, program execution falls through to `NEXT` and starts another loop (if `J` isn’t bigger than 10). If `NUMBER(J)` is smaller than `TEST`, then that particular value of `NUMBER(J)` is assigned to variable `TEST`, causing `TEST` to have a value equal to the smallest value of `NUMBER(J)` encountered so far. By the time all elements of the array `NUMBER` have been checked in this manner (this happens with the last loop when `J = 10`), `TEST` will have a value equal to the smallest member of the list of numbers in the array. Voila!

Starting Over with `ERASE`

An array-related statement that may occasionally be important is the `ERASE` statement. Its execution causes an array to be “erased”: it removes the array’s content and frees the memory space initially reserved by the `DIM`

statement. The statement consists of the word *ERASE*, followed by a list of one or more array names, as in this example:

```
1000 ERASE MONEY(100), ANIMAL$(25,12)
```

which ERASEs the two listed arrays.

There are several reasons to use ERASE. If we no longer need a large array that takes up needed memory, we can “reclaim” the memory space initially reserved by a DIM statement by means of the ERASE statement. Also, we occasionally need to redimension an array. We cannot do it, without getting an error message, simply by executing another DIM statement; we must first use ERASE to eliminate the memory space created by the first DIM statement.

Summary

In this chapter we’ve covered two ways to handle large amounts of information. The first allows us to store and READ a collection of DATA in a compact and efficient manner. DATA and READ statements are especially useful for initializing variables or arrays with large blocks of fixed information. The second technique involves arrays, the elements of which are referred to by array names and subscripts. Arrays are ideal for temporarily storing blocks of information and for manipulating (rearranging, adding, and so on) information of the same type.



Exercises

1. Write a program that stores the names of the notes DO, RE, MI, and so forth in an array and asks the user to enter a note number (between 1 and 8); the program should then print out the name of the note.
2. Write a program that generates a five-by-five multiplication table. Use a two-dimensional array to store the products of row and column numbers and then print out the array. Also, find the sum of the numbers located on the *diagonal* of the multiplication table, beginning with the upper left product and ending with the lower right product.

Solutions

1. The following program prints out the name of a note, given a note number supplied by the user:

```
10 REM--NAME:NTENUM.BA-----
20 REM--prgm requests user input of
30 REM--note number & prints out name
40 REM--of note
50 '
110 DIM NTE$(8)
120 DATA DO, RE, MI, FA, LA, SO, TI, DO
130 FOR J = 1 TO 8 'initializes array
140     READ NTE$(J)
150 NEXT
160 '
170 INPUT;"Note number (1-8)", J
180 PRINT "  the note is " NTE$(J)
190 GOTO 170
200 END
RUN
Note number (1-8)? 3
    the note is MI
Note number (1-8)? 6
    the note is SO
Break in 70
OK
```

2. The following program prints out a five-by-five multiplication table as well as the sum of the diagonal entries in the table:

```
10 REM--NAME:MULTBL.BA-----
20 REM--prgm prints 5*5 mult. table
30 REM--and sum of diag. elements
40 '
100 CLS
110 MX = 5                                     ← Specifies size of array N
120 DIM N(MX,MX)
130 '
134 '--mult. table stored in array N--
136 '
140 FOR ROW = 1 TO MX
150     FOR COL = 1 TO MX
160         N(ROW,COL)=ROW*COL                 ← Loads array element with
170     NEXT COL                               ← product ROW*COL
180 NEXT ROW
190 '
194 '--prints out array N-----
196 '

```

```

200 FOR ROW = 1 TO MX
210   FOR COL = 1 TO MX
220     P=(COL*4-1)+(ROW-1)*40
230     PRINT @P, N(ROW,COL)
240   NEXT
250 NEXT
260 '
270 '--finds sum of diag. elements----
280 '
290 SUM = 0
300 FOR DIAG = 1 TO MX
310   SUM = SUM + N(DIAG,DIAG)
320 NEXT
330 PRINT @ "Sum of diag. elements = "; SUM
500 END

```

← Defines screen coord. P
 ← Prints array element at P
 ← Accumulates sum of diagonal

When this program is RUN, the Model 100 displays the following multiplication table:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25
Sum of diag. elements = 55				
OK				

Each number in this table is the product of its row and column number. For example, 15 in the fifth row and third column is the product of 5 (the row number) and 3 (the column number).

Notice the use of nested FOR...NEXT loops to initialize as well as print the array N(ROW,COL). The inner loop controls the columns, the outer loop, and the rows. To find the sum of the diagonal elements, we used the following standard summing procedure: add the old value of SUM to the next diagonal value, N(DIAG,DIAG), and assign this updated sum to the variable SUM. The initial value of SUM is set to 0 by line 290, and the last value is the sum of all the diagonal elements. This procedure is similar to the way we add numbers with a calculator: we begin with an initial sum of 0, and each time we add a new number to the old subtotal, our calculator displays the new subtotal; the last subtotal displayed is the final answer — the sum of all the values that we entered.

Dot Graphics

Concepts

Drawing and erasing dots
Drawing and erasing lines and boxes
Patterns and graphs

Instructions

PSET, PRESET, LINE

The Model 100 has sophisticated graphics capabilities that enable us to draw interesting and useful images on the screen. We've already introduced *character graphics*, which enables us to place a character on any of the 320 possible character positions on the screen (40 columns by 8 rows). In this chapter we explore *dot graphics* (sometimes called *all-points-addressable graphics*). With this type of graphics, we'll be able to draw intricate patterns of dots and lines, as well as various detailed graphs.

To understand what dot graphics means, we need to take a closer look at the Model 100 screen, which is made up of a large array of small squares called *pixels*. Each pixel can be either "on" or "off"; when a pixel is turned on, it forms a dark dot. All characters we've used so far and all the graphics images we'll show you in this chapter are constructed from this array of pixels.

Dot graphics enables us to individually turn on and off any of the 15,360 pixels (240 in the horizontal direction by 64 in the vertical) that make up the Model 100 screen. Because there are many more pixels than character positions (to be exact, forty-eight times as many), dot graphics can produce much finer, more detailed images than character graphics. In short, dot graphics has a much higher *resolution* — the ability to resolve detail — than character graphics. In addition, dot graphics on the Model 100 makes it extremely easy to draw lines and boxes.

Putting Points on the Screen with PSET

PSET is the simplest of all the statements related to dot graphics. *PSET* stands for “Point-SET”: it places or “sets” a “point” by turning on a pixel at a specified location. Let’s try the following direct command after clearing the screen with CLS:

```
PSET (120,32)
```

A small dot appears at the center of the screen. This command says, “Set a point at the location given by the two numbers in parentheses”. To find out what these numbers, or *coordinates*, have to do with the location of the spot, try changing the first number; for example, try the following command:

```
PSET (140,32)
```

The result is a point to the right of the original one. So the first number determines the *horizontal* position of the point: the bigger the number, the farther to the right is the point’s location. This horizontal coordinate is frequently referred to as the *x-coordinate*. A similar experiment will reveal that the second number — usually called the *y-coordinate* — defines the *vertical* location of the point: the larger the y-coordinate, the lower the screen position. This may seem a bit peculiar to anyone with some experience in making graphs — a larger y-coordinate usually means a higher position — but it does agree with our sense of row numbers increasing as we go down the page. Figure 13-1 summarizes how coordinates are specified.

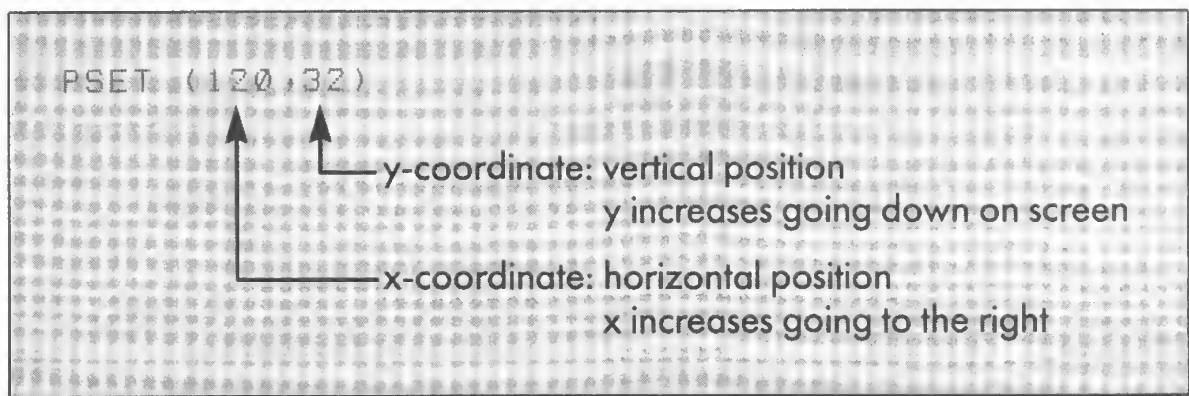


Figure 13-1. The PSET statement

Now enter the following program, which demonstrates the extent of the screen in terms of the coordinates x and y:

```
10 CLS
20 PSET (0,0)           ← Upper left corner
30 PSET (239,0)         ← Upper right corner
40 PSET (0,63)          ← Lower left corner
50 PSET (239,63)        ← Lower right corner
60 PRINT @80, " "
60 END
```

When you RUN this program, four small dots appear near the corners of the screen, defining its limits. The coordinates of these points (the corners) are illustrated in Figure 13-2. The numbers in parentheses are the x- and y-coordinates of the screen's corner points, the same numbers that appear in the PSET statements. The upper left corner, defined by both x and y equal to zero (written as (0,0) is called the *origin*; it's the place from which we start counting. The largest x value is 239, which means that the screen has a total of 240 horizontal or x-positions (239 plus 1 for the zero position). Similarly, the largest value of y is 63, to give us a total of 64 vertical positions.

Some Examples Using PSET

PSET is extremely useful for producing graphics output. We can construct any graphics image by using a combination of PSET statements. As in our previous example, we can use one PSET statement to turn on one particular pixel, though it would be rather tedious to assemble a complex

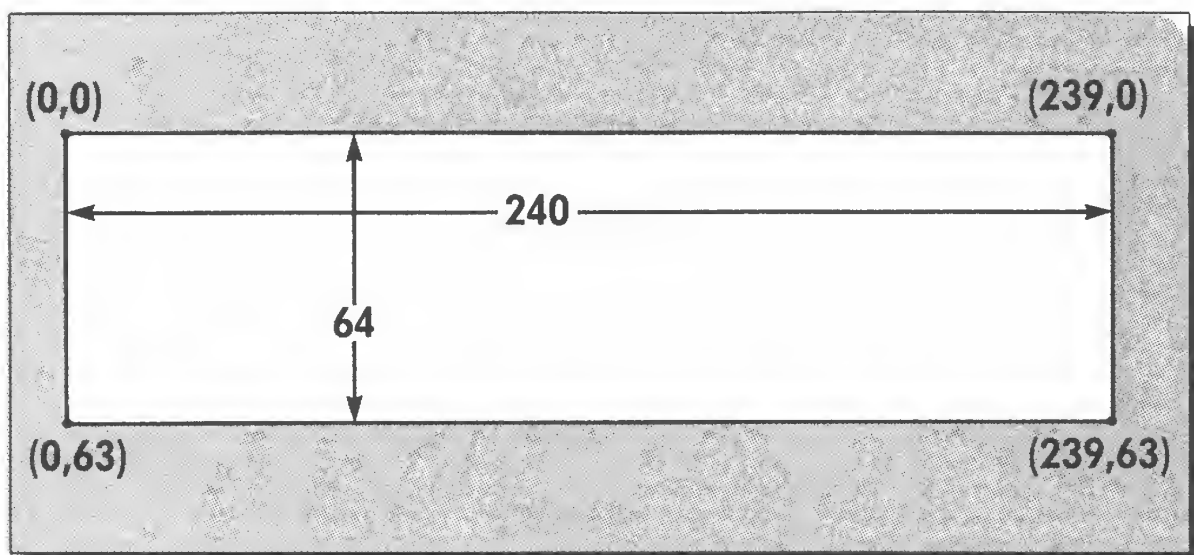


Figure 13-2. Limits of screen coordinates

image in this manner. We can usually get around this problem by using the FOR...NEXT loop to draw a sequence of dots with the same PSET statement.

For example, let's write a program that draws a line of dots that starts at the upper left corner of the screen and has an angle of forty-five degrees:

```
10 CLS
20 FOR J = 0 TO 63
30   PSET (J,J)
40 NEXT
```

The first execution of the loop places a dot at (0,0), the second at (1,1), and so forth until $J = 63$. As you can see, the x- and y-coordinates specified inside the PSET statement can be variables.

Now let's change our program so that it draws a diagonal across the screen. Here is one way to do it:

```
10 CLS
20 FOR J = 0 TO 239
30   PSET (J, J/3.75)
40 NEXT
```

This program places 240 dots, one for each possible x-position. The first loop places a dot at the coordinates (0,0), and the last loop places a dot at the lower right corner, coordinates (239,63). Notice that the coordinates within the PSET statements can be specified as expressions, such as $J/3.75$ in our example. Furthermore, such expressions don't need to translate to integers. For example, the expression for the y-coordinate within our PSET statement ($J/3.75$) translates to 63.73 (approximately) for $J = 239$. BASIC on the Model 100 *truncates* this number to give 63, the maximum y-coordinate on the screen.

Once we know how to use PSET to draw a line, it's easy and fun to invent programs that do all sorts of interesting things. The following is a simple example that draws a zigzag line that looks like the track of a billiard ball bouncing off two sides of an endless billiard table:

```
100 REM--NAME:ZIGZAG.BA-----
110 REM--Prgm draws zigzag line between
120 REM--coordinates Y1 and Y2
130 REM
140 DEFINT A-Z
```

```

200 Y1 = 20      'upper limit in y
210 Y2 = 40      'lower limit in y
220 DX = 2       'x increment
240 DY = 2       'y increment
250 X = 0        'beginning x-coord.
260 Y = Y2       'beginning y-coord.
270 FOR J = 0 TO 239/DX
280   IF Y>=Y2 OR Y<=Y1 THEN DY=-DY
290   PSET (X,Y)
300   X = X + DX
310   Y = Y + DY
320 NEXT
330 END

```

← Reverses direction at
 ← boundaries Y1 and Y2
 ← Advances dot in x direction
 ← Advances dot in y direction

Lines 300 and 310 update the old values of x and y by adding the values of DX and DY, the amount that the x- and y-coordinates change with each execution of the FOR...NEXT loop. DX has a constant value of 2 throughout the whole program, whereas DY switches back and forth between 2 and -2 as directed by the IF...THEN statement in line 280. That's how we get the line to zigzag. The output is shown in Figure 13-3.

It would be easy to modify this program so that the “ball” bounces off four sides rather than just the two used in this program. All it takes is another IF...THEN statement that limits X in the same way that the existing IF...THEN statement limits Y. (If you like this program, you'll enjoy our solution to exercise 1 at the end of this chapter.)

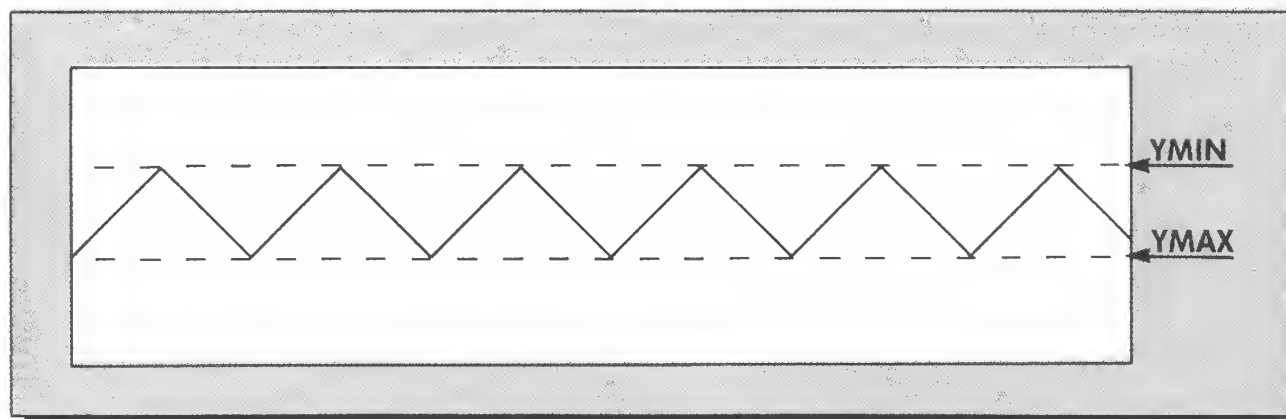


Figure 13-3. Output of program “ZIGZAG.BA

Drawing a Simple Graph with PSET

Graphs that show the relationship between two quantities are a more serious and practical use of graphics, and in particular of the PSET statement. For example, we might want to “see” how an investment grows with time. This is a fairly complicated programming problem, so we’ll save it for the exercise section at the end of this chapter. We’ll start out with the simpler problem of graphing the square of a number versus the number itself. The x-coordinate on the Model 100 screen is to be proportional to the number N , and $N = 0$ is to be located at the screen coordinate $x = 120$ (halfway across the screen). The y-coordinate is to be proportional to the square of the number, $N*N$. Here is our program:

```
100 REM--NAME:SQGRPH.BA-----
110 REM--prgm graphs number N versus
120 REM--its SQUARE
130 REM
200 CLS
205 DEFINT A-Z
210 FOR N = -25 TO 25
220   SQUARE = N*N
230   X = N*4 + 120           ← Converts N to screen coordinate X
240   Y = 63 - SQUARE/10     ← Converts SQUARE to screen coordinate Y
250   PSET (X,Y)
260 NEXT
270 END
```

The output of this program, as shown in Figure 13-4, has the shape of a parabola (the same shape used in telescope mirrors to focus light).

The program is easy to understand, although there is one aspect relevant to all graphing programs that needs some explanation: the problem of translating the values to be graphed to the screen coordinates x and y . In

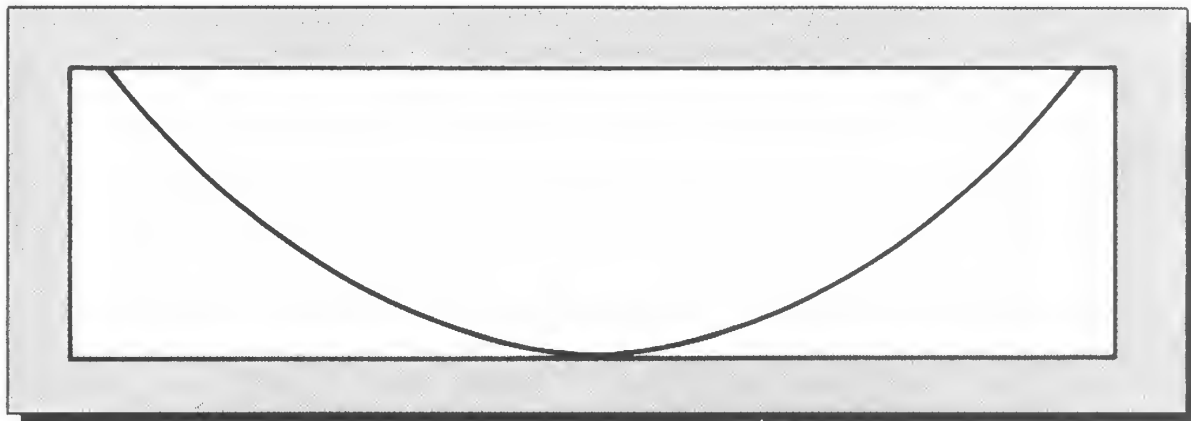


Figure 13-4. Output of program “SQGRAPH.BA.”

this example, we must translate the number N into the x-coordinate of the screen and the square of N , $SQUARE$, into the y-coordinate of the screen. These translations are made by lines 230 and 240. Line 230 multiplies the value of N by 4 and then adds 120 to give us the screen coordinate x . The purpose of adding 120 is to place the zero x value in the center of the screen (in the horizontal direction); the purpose of multiplying N by 4 is to horizontally “stretch out” the graph to use the full dimensions of the screen. Line 240 is slightly more complicated. First, the value of $SQUARE$ is divided by 10 because we must confine our screen y-coordinates to the range 0-63. The largest value of $SQUARE$ is equal to 625, which is clearly too large to be a y-coordinate. Second, the quotient $SQUARE/10$ is subtracted from 63. If we omitted this step, our graph would be upside down. The subtraction causes larger values of $SQUARE$ to be located at *higher* positions on the screen. The particular value of 63 was chosen so that when $SQUARE$ equals 0, a dot will be placed at the lower edge of the screen.

Our program draws a very nice parabolic curve. Notice, however, that although you can get a good visual sense of the curve, you can’t tell what the actual values are. You can’t tell from the graph, for example, that the square of five is twenty-five. What we need are *axes* and *tick marks* that tell us how to translate a distance on an axis into a numeric value. We’ll show you how to deal with this problem in our solution to exercise 3 at the end of this chapter.

The PSET Statement

PSET places a point on the screen at the x- and y-coordinates specified in the following manner:

PSET (X,Y)
 ↑ ↑
 | |
 | | y-coordinate (from 0 to 63)
 | |
 | | x-coordinate (from 0 to 239)

For example, PSET (120,32) places a point at the center of the screen. The coordinates x and y may be numbers, variables, or expressions. Coordinates having decimal values are truncated to integers.

Erasing Dots with PRESET

You now know how to use PSET to place dots anywhere on the screen. But some applications also require us to *erase* dots. For example, for a moving line segment, dots must be added in “front” of the segment and trailing dots must be erased. The statement that erases dots (or turns pixels off) is the PRESET statement.

PRESET is identical to PSET except that whereas PSET turns a pixel on, PRESET turns it off; *PRESET* stands for “Point RESET”. Screen coordinates in a PRESET statement are specified in the same way as in a PSET statement.

Animating Dots

A program that causes a dot to *move* across the screen is an interesting application of PRESET. For example, the following program moves a dot across the screen from left to right:

```
10 REM--dot moves from left to right---
20 REM
30 FOR J=5 TO 239
40   PSET (J, 32)           ← Places dot
50   FOR P=1 TO 20: NEXT    ← Time delay
60   PRESET (J,32)          ← Erases dot
70 NEXT J
80 END
```

PSET places a dot and PRESET erases the same dot after a short time delay. With each execution of the FOR...NEXT loop the dot advances one pixel to the right, because the x-coordinate in both PSET and PRESET is the loop index J.

A variation on the same theme is to move a *line segment* across the screen. We need only replace line 60 in our last program with the following line:

```
60   PRESET (J-5, 32)
```

When RUN, this modified program will move a line segment five dots long from left to right. If we want to speed up the motion, we can remove the delay statement in line 50 without significantly sacrificing visibility.

The PRESET Statement

The PRESET statement *erases* a dot (turns off a pixel) at a specified coordinate, as shown in the following example:

```
150 PRESET (120,32)
```

This statement turns off a pixel at the x-coordinate equal to 120 and y-coordinate equal to 32.

Drawing Lines and Boxes with LINE

Though we can construct lines from individual points with PSET, Model 100 BASIC provides us with a single instruction designed for just this purpose — the LINE statement. It's a powerful and versatile statement. In addition to drawing lines, the LINE statement can also be used to draw both empty and filled boxes and to erase lines and boxes. As an introduction to LINE, enter the following direct command:

```
LINE (0,0) - (239,63)
```

A diagonal line will appear on your screen from the upper left corner to the lower right corner, as shown in Figure 13-5.

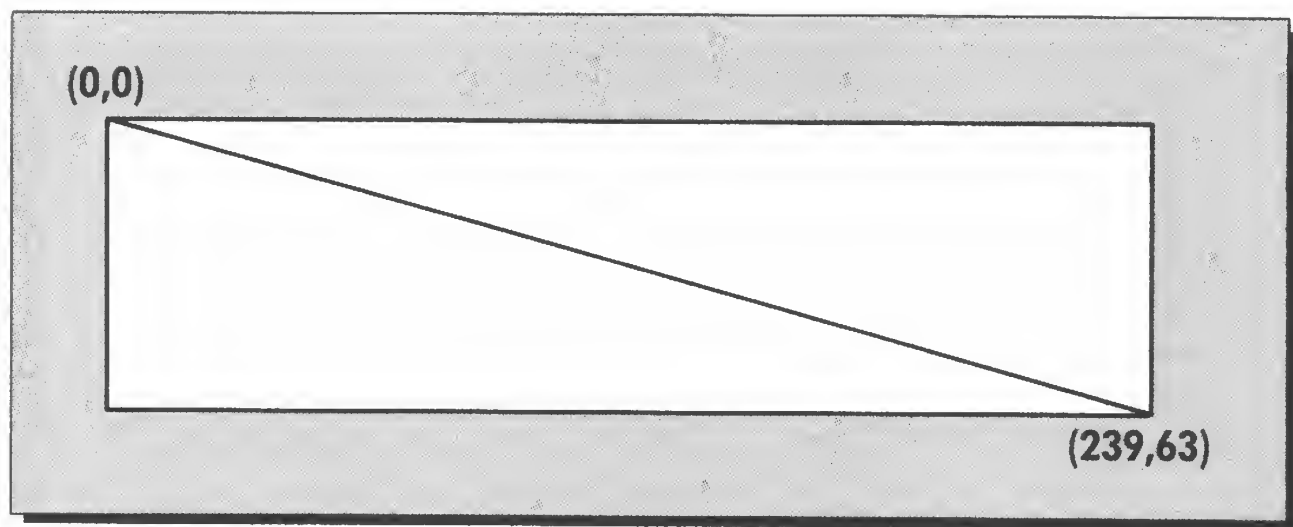


Figure 13-5. A line drawn by LINE (0,0) - (239,63)

This command draws the line much faster than PSET within a FOR...NEXT loop does. This happens because this line is drawn by a routine written in machine language, whereas a line drawn using repetitions of PSET requires the execution of many separate BASIC statements, which takes much longer.

The end points of the line drawn with LINE are specified by the two sets of coordinates listed after LINE, as shown below:

```
LINE (0,0) - (239,63)
```

x- and y-coordinates of first end point

x- and y-coordinates of second end point

LINE draws a line from the first set of x- and y-coordinates to the second. In our example, the line was drawn from (0,0), the upper left corner, to (239,63), the lower right corner. As you can see, coordinates are specified in LINE in the same way they are specified in PSET, except that LINE requires *two* sets of coordinates to define the two end points of the line. Incidentally, we've written the above LINE statement with spaces at various points to make it easier to read, but BASIC really doesn't care about them: we could write the statement without any spaces at all.

The following example uses LINE to draw an interesting pattern:

```
10 REM--NAME: LINES1.BA-----
15 DEFINT J,X,Y
20 CLS
30 FOR J = 0 TO 20
40   X = 50 + J*8
50   Y = 3 + J*3
60   LINE (50,Y) - (X,63)
70 NEXT
80 END
```

The output of this program is shown in Figure 13-6. The program draws twenty lines that create a pattern similar to one you might see in the familiar pin and string pictures. Note that, as with PSET, the coordinates in LINE can be variables or even numeric expressions.

Erasing with LINE

We can also use the LINE statement to *erase* a line by using an additional number or variable. Unlike PSET and PRESET, which form a pair of statements that print or erase a dot, LINE can handle both functions — drawing and erasing — by itself.

To see how we can use LINE to erase a line, clear the screen and enter the direct command

```
LINE (40,32) - (200,32)
```

to draw a horizontal line in the center of your screen. Now enter the following direct command:

```
LINE (40,32) - (200,32),0
```

The line drawn by the first command disappears! Notice that the only difference between the two commands is the comma and the zero (0) appended to the second command — the zero causes the LINE instruction to *erase* rather than draw a line.

But that's only part of the story. We used a zero at the very end of the second command to tell LINE to erase rather than draw. It turns out, however, that *any even number* has the same effect. That is, 0, 2, 4, 6, 8, and so forth all cause LINE to erase when placed after the second coordinate of a LINE statement. Can we use *odd* numbers as well? Yes, but they have the same effect as no number at all; that is, odd numbers cause LINE to *draw* a line rather than erase it.

Although using a number as a switch to determine whether LINE draws or erases a line may seem unnecessarily complex, it gives the LINE state-

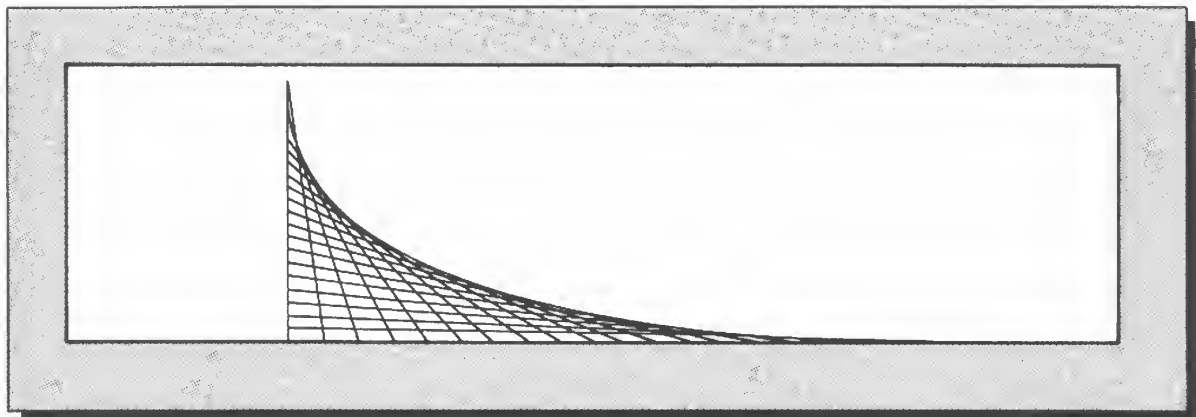


Figure 13-6. Out of program "LINE51.BA"

ment a flexibility it wouldn't otherwise have. For example, consider the following program, which draws a blinking line:

```
10 CLS
20 FOR SW% = 1 TO 16
30   LINE (40,32) - (200,32),SW%
40   FOR T = 1 TO 100: NEXT
50 NEXT
```

Notice that our switch in `LINE` is the *variable* `SW%`. Because `SW%` is also the loop index, the value of the switch variable (`SW%`) alternates between odd and even with each execution of the `FOR...NEXT` loop. When `SW%` is odd, `LINE` draws a line; when `SW%` is even, `LINE` erases the line. The function of line 40 is to slow down the blinking rate.

Finally, let's modify an earlier program, "LINES1.BA", so that the screen image is erased in the same manner as it was drawn. Try it out — the program creates a visually interesting effect that may give you some ideas for your own programs.

```
100 REM--NAME:LINES2.BA-----
110 REM--draws and then erases series
120 REM--of lines
125 DEFINT A-Z
130 CLS
135 SW = 1
140 FOR J = 0 TO 20
150   X = 50 + J*8
160   Y = 3 + J*3
170   LINE (50,Y) - (X,63),SW
180 NEXT
190 SW = SW + 1
200 GOTO 140
210 END
```

The new feature in this program is the switch variable `SW` in the `LINE` statement. The first time the whole set of `FOR...NEXT` loops is executed, `SW` has the value 1, causing the lines to be drawn (instead of being erased). However, after the whole set of lines is drawn, line 190 increments `SW` by 1 to give the value 2. The `GOTO` statement in line 200 then directs program execution back to the beginning of the `FOR...NEXT` loop. The result is that another set of `LINE` statements is executed, *except* that now the switch value is the number 2 — which is *even* — so each `LINE` statement now erases a line drawn previously. The program keeps going indefinitely — drawing a set of lines, then erasing them. In the exercises, we'll expand this program to make it even more interesting.

The LINE Statement (Without the Box Option)

The LINE statement draws a line from the first to the second set of x- and y-coordinates. An optional number or variable may append the LINE statement to determine whether a line is to be drawn or erased. For example, the statement

Optional switch number or variable:
if odd, causes line to be drawn;
if even, causes line to be erased

100 LINE (100,10) - (160,40),SW

First end point of line

Second end point of line

causes a line to be drawn from the screen coordinates (100,10) to (160,40) if the value of SW is odd or if SW and the comma that precedes it are omitted altogether. If SW is even, the specified line is erased.

Drawing Boxes with LINE

Because a rectangle has four sides, it takes four LINE statements to construct a rectangle, right? Only if you want to do it the long way. BASIC on the Model 100 provides a shortcut. You need only amend the LINE statement with a *B*, as illustrated in the following example:

```
LINE (100,12) - (140,52),1,B
```

A pretty compact instruction! Figure 13-7 shows the box drawn by this command. The two sets of x- and y-coordinates specified in the above LINE command determine the location of two opposite *corners* of the box. The letter *B* at the end of the command causes LINE to draw a box rather than a line. Note also that we included the number 1 right before the letter *B* to instruct LINE to draw rather than erase (odd numbers cause LINE to draw, even, to erase). Whereas this switch value can be omitted in LINE without the *B*, it is *required* if the *B* option is used, whether you want to draw or erase a box.

Want to see an interesting use of the *B* option? Simply add the letter *B*

to the end of line 170 in the previous program. With this simple change, every line in the program is now drawn as a box!

Drawing boxes with LINE is useful for framing titles or graphs, creating bargraphs, visually organizing material on the screen, or simply creating interesting visual patterns. The following program illustrates how LINE might be used to create a set of boxes that frame a message:

```

100 REM--NAME: BOXES1.BA-----
110 REM--prgm draws 10 concentric boxes
120 REM--around a message
130 REM
135 DEFINT A-Z
140 CLS
150 XC = 122
160 YC = 35
170 WID = 106
180 HGHT = 19
190 FOR J = 1 TO 10
200   X1 = XC - WID/2
210   X2 = XC + WID/2
220   Y1 = YC - HGHT/2
230   Y2 = YC + HGHT/2
240   LINE (X1,Y1)-(X2,Y2),1,B
250   WID = WID + 6
260   HGHT = HGHT + 4
270 NEXT
300 PRINT @173, "HAVE A NICE DAY"
310 END

```

- ← X-coordinate of center of box
- ← Y-coordinate of center of box
- ← Width of box
- ← Height of box
- ← Draws ten boxes
- ← X-coord. of upper left corner
- ← X-coord. of lower right corner
- ← Y-coord. of upper left corner
- ← Y-coord. of lower right corner
- ← Draws one box
- ← Increments width
- ← Increments height

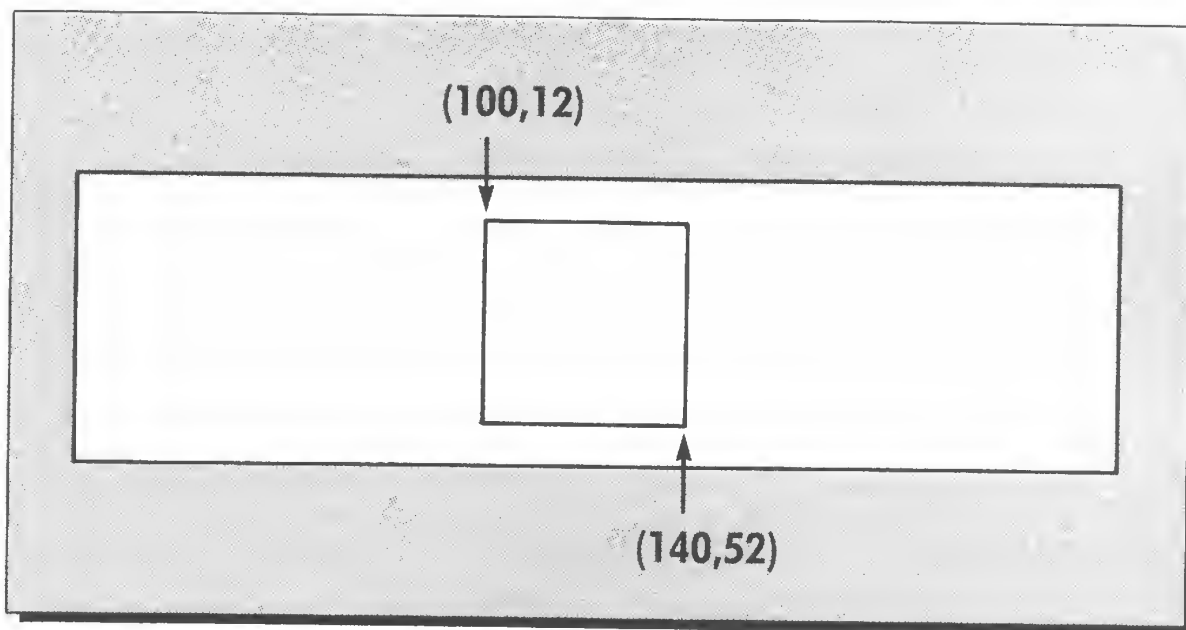


Figure 13-7. Box drawn by LINE (100,12) – (140,52), 1, B

We can easily adapt this program to draw *any* box or sequence of boxes of varying widths, heights, and center coordinates. For example, changing the value of XC would move the whole set of boxes horizontally. Note that although we could write this program more compactly, defining intermediate variables such as XC and WID makes it easy to understand and modify.

Filling Boxes with LINE

For applications such as bar graphs, we may wish to use solid or filled boxes — boxes filled with turned-on pixels. BASIC for the Model 100 again supplies an easy solution. We just append the letter *B* at the end of the LINE statement with an *F*, for “Filled”. For example, the statement

```
LINE (100,12) - (140,52),1,BF
```

produces a solid or filled box having the same corner positions as the previous box.

LINE with the *B* and *BF* Options

The following LINE statement

```
100 LINE (0,0) - (239,32),1,B
```

x- and y-coordinates of corners of box

↑ Causes box to be drawn (if a BF is used a filled box is drawn)
↑ This number, or any odd number, is required to draw a box (even numbers erase the box)

draws a box. As the arrows demonstrate, the two sets of x- and y-coordinates specify the position of two opposite corners of the box. The *B* option draws an empty box, the *BF* option, a filled box.

Summary

Doing graphics is one of the most enjoyable aspects of programming. In this chapter we have introduced three statements related to *dot graphics*: the PSET statement, which *places* a dot at a given x- and y-coordinate; the PRESET statement, which *erases* a dot at a given coordinate; and, finally, the LINE statement, which can be used to draw and erase lines, as well as to draw empty and filled boxes.

We can use these statements to draw a vast variety of patterns, as well as to draw graphs that can be used in business, private finance, or mathematics class! We have presented fairly simple examples to illustrate the basic concepts, but be sure to look over the examples in the exercises. These illustrate more complex uses of the graphics statements explained in this chapter.

In the next chapter, we introduce arithmetic functions on the Model 100. You'll find some of these, especially the trigonometric functions, very useful in extending the graphics capabilities of your Model 100.



Exercises

1. Modify the program ZIGZAG.BA (presented earlier) so that a dot seems to bounce off *four* walls drawn by a LINE statement. The moving dot should leave a trail behind it to create an interesting pattern.

2. Modify the program LINES2.BA so that the program draws lines all the way around a rectangle. There should be four LINE statements, each of which draws a line between two adjacent sides of the rectangle, as done in LINES2.BA. The program should also erase all the lines drawn as soon as the pattern is complete.

3. Write a program that graphs the present value of an investment earning a fixed interest rate, compounded monthly. The amount initially invested should be represented by the value 1 (which can represent \$1,000, \$100,000, or whatever amount you wish), and the total investment period should be eight years.

Solutions

1. Here is our solution to the moving dot problem:

```
100 REM--NAME:BILLRD.BA-----
110 REM--dot with trail bounces off
120 REM--four walls of rectangular box
130 REM
135 DEFINT A-Z
140 CLS
145 '
146 '---initialize variables-----
147 '
150 X1 = 80: Y1 = 10
160 X2 = 160: Y2 = 52
170 DX=1: DY=1
180 X = X1 + 1
190 Y = Y1 + 15
195 '
196 '---draw box in which dot bounces
197 '
200 LINE (X1,Y1) - (X2,Y2),1,B
205 '
206 '---bouncing dot with trail-----
207 '
210 FOR J = 0 TO 1000
220 IF Y>=Y2 OR Y<=Y1 THEN DY = -DY
230 IF X>=X2 OR X<=X1 THEN DX = -DX
240 PSET (X,Y)
250 X = X + DX
250 Y = Y + DY
260 NEXT
270 END
```

← One corner of box
← Opposite corner of box
← Specifies 'velocity' of dot
← Specifies initial x of dot
← Specifies initial y of dot

← Reverses y-direction at wall
← Reverses x-direction at wall

← Advances dot in x direction
← Advances dot in y direction

s 76

2. This program draws and erases a rectangular string pattern:

```
100 REM--NAME:LINE$3.BA-----
110 REM--draws lines to adjacent sides
120 REM--of rectangle
130 REM
135 DEFINT A-Z
140 CLS
150 SW = 1 'initial switch: draw lines
```

```

155 '
156 '---draw lines-----
157 '
160 FOR J = 1 TO 20
170   DX = J*8
180   DY = J*3
190   LINE (40,DY)-(40+DX,63),SW
200   LINE (40+DX,63)-(200,63-DY),SW
210   LINE (200,63-DY)-(200-DX,0),SW
220   LINE (200-DX,0)-(40,DY),SW
230 NEXT
235 '
236 '---repeat pattern, erase or draw
237 '
240 SW = SW + 1 'increments switch
250 GOTO 160
300 END

```

3. This program graphs the present value (PV) of an investment earning a fixed interest, compounded monthly:

```

100 REM--NAME:EXGTH2,BA-----
110 REM--PV, initially=1 is graphed
120 REM--vertically; time for up to
130 REM--8 years is horizontal axis,
140 REM
150 CLS
155 '
156 '--info. to user and interest input
157 '
160 PRINT TAB(6) "GRAPHING EXPONENTIAL GROWTH"
170 PRINT
180 PRINT "Vertical tick marks represent
190 PRINT "  present value (PV) from 0 to 6"
200 PRINT
210 PRINT "Horiz. ticks are yrs. from 0 to 8"
220 PRINT
230 INPUT " Interest(%)"; IP
240 CLS
245 '
246 '--initialize variables & call SUBR
247 '
250 PV = 1 'initial investment
250 I = IP/1200 'monthly fractional interest rate
255 '

```

```

260 '=====  

270 GOSUB 1000 'draws axes and ticks  

280 '=====  

285 '  

286 '--finds PV and graphs it-----  

287 '  

290 FOR MO = 1 TO 96  

300   PV = PV*(1 + I)          ← Updates Present Value, PV  

310   X = 45 + MO*2           ← Translates MO into screen x  

320   Y = 63.5 - PV*10       ← Translates PV into screen y  

330   IF Y<3 THEN 990        ← If out-of-bounds, then END  

340   PSET (X,Y)  

350 NEXT  

990 END  

995 '  

1000 '===SUBR. for axes and ticks===  

1005 '  

1010 LINE(45,63)-(237,3),1,B 'frame  

1020 '  

1030 '--draw vertical ticks-----  

1040 '  

1050 FOR J = 0 TO 6  

1060   Y = 63 - J*10  

1070   LINE (237,Y)-(235,Y) 'right side  

1080   LINE (45,Y)-(47,Y)  'left side  

1090 NEXT  

1100 '  

1110 '--draw horizontal ticks-----  

1120 '  

1130 FOR J = 0 TO 8  

1140   X = 45 + 24*J  

1150   LINE (X,63)-(X,61)  

1160 NEXT  

1170 '  

1180 '--Print value of interest rate---  

1190 '  

1200 PRINT @0, "Int, ="  

1210 PRINT @B1, IP; "%";  

1300 RETURN

```

Although this is a long program, you'll find it easy to follow. Try running the program; it shows very graphically the effects of compounding interest: as time goes on, the curve showing the present value (PV) of your investment gets steeper and steeper. You can also use this program to ask questions such as: "How long will it take me to double my money at 18 percent interest?". This type of growth is often referred to as "exponential growth". Many quantities other than savings at fixed rates tend to increase in this manner.

Numeric Functions

Concepts

Functions
Sign-related functions
Trigonometric functions
Powers and exponential functions
Random numbers

Instructions

ABS, SGN, CINT, FIX, INT, TAN, SIN, COS, ATN, EXP, LOG, RND

You've seen how to program your Model 100 to perform a wide variety of tasks that range from making decisions to graphing pictures and investment returns. Many of the programs we've written involve the basic arithmetic operations of addition, subtraction, multiplication, and division. But the Model 100 has mathematical abilities far beyond these four operations. Some of these mathematical "talents" are used primarily for scientific applications, but many are of general interest, particularly in business and graphics applications. In this chapter we'll explore the different ways the Model 100 can manipulate numbers by means of *numeric functions*. Computers were invented to do just this type of numeric calculation, and it's still one of the things that computers do best.

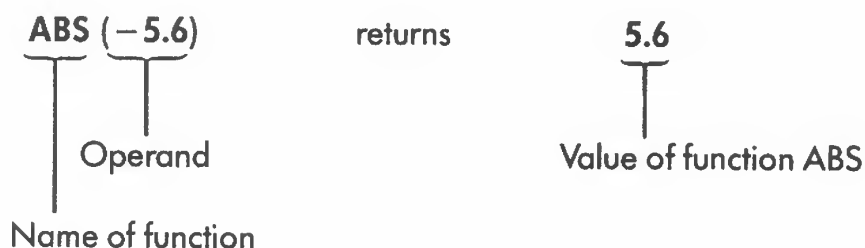
Don't think that you need to be a mathematician to read this chapter — far from it. Though some previous exposure to algebra and trigonometry is beneficial, it is not essential in order to learn from and enjoy this chapter. In fact, we find that the computer is a wonderful tool for teaching (and reviewing) many basic mathematical concepts. If you really just can't stand mathematics, however, you may wish to skip this chapter. Nothing here is essential to your understanding of the remaining chapters.

Numeric Functions — An Introduction Using ABS and SGN

We'd like to introduce the concept of numeric functions with a specific example. The ABS function is ideal for this purpose because it's very simple. To begin, enter the following direct commands:

```
ABS(5.68)
5.68
OK
ABS(-2.5)
2.5
OK
```

As you can see, the output is the same as the number inside the parentheses after ABS, except that the negative sign is gone. *ABS* stands for “ABSolute value”; it is the name of a function that takes a number and returns the same number but drops the negative sign if there is one. The absolute value of any number is always positive. The number inside parentheses after ABS is called the *operand* — it's the number that the function ABS “operates” on to produce the value of the function. We can summarize the various aspects of this function as follows:



So far, we've used a number as the operand, but the operand can be a variable or a numeric expression. Also, the value of the function ABS can be assigned to another variable. The following example illustrates these variations:

```
10 INPUT N
20 MAGNITUDE = ABS(N)
30 PRINT N, MAGNITUDE
```

The ABS is typical of most numeric functions: a *numeric function* takes a number (the operand) and returns another number (the value of the function), which is determined by rules dictated by the particular function. In the case of the ABS function, the rule is simply to “drop all negative signs”. Figure 14-1 summarizes what a function does.

The SGN Function

Another sign-related function is SGN. *SGN* stands for “SiGN”, and it tells us the sign of a number. As an example, run the following program:

```
10 FOR N = -3 TO 2
20   PRINT N, SGN(N)
30 NEXT
OK
RUN
-3      -1
-2      -1
-1      -1
 0       0
 1       1
 2       1
OK
```

As you can see, SGN(N) returns 1 when N is positive, 0 when N is 0, and -1 when N is negative. Figure 14-2 summarizes how SGN works.

Making Integers with CINT, FIX, and INT

Some applications require that a program convert a decimal number (single or double precision) to an integer value. (Later we’ll present an example that converts a decimal time value to the equivalent in hours, minutes, and seconds.) Although converting a decimal number to an integer value is a simple process, there are, nevertheless, three different functions that make this conversion — the CINT, FIX, and INT functions.

Let’s consider the CINT function first. Try the following direct command.

```
PRINT CINT(3.55)
3
OK
```

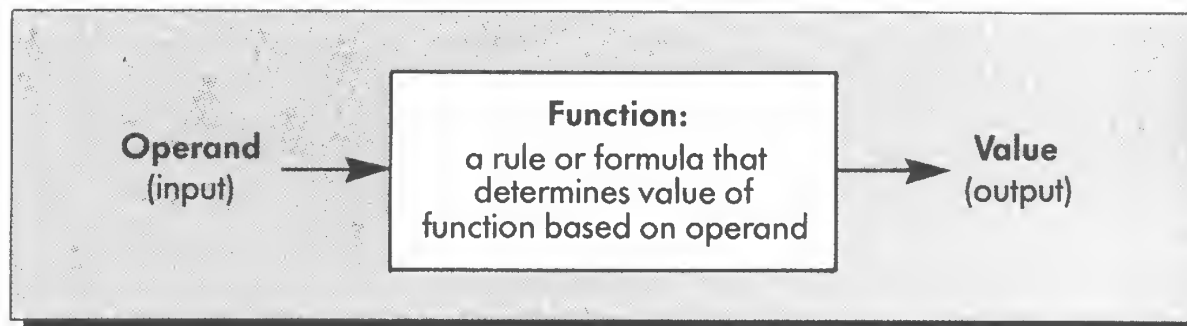


Figure 14-1. Anatomy of a numeric function

Given an operand equal to 3.55, CINT returns the value 3, which is the integer part of the operand. That is, CINT simply drops or “truncates” the decimal part of the operand. *CINT* stands for “Convert to INTegeR”: this function takes a single- or double-precision number and converts it to an integer value requiring only 2 bytes of memory. This means that the value of CINT must be in the usual range of integer values — namely, between -32,768 and 32,767.

The two other BASIC instructions that do a job similar to CINT are FIX and INT. For *positive* operands, FIX and INT appear to have exactly the same effect as CINT. For example, enter the following:

```
PRINT FIX(5.9); INT(5.9)
5 5
OK
```

Both FIX and INT truncate the operand 5.9 to return the value 5 — exactly what CINT would have done. The difference, however, between CINT and the functions FIX and INT is that whereas CINT returns a true integer (two bytes of memory), both FIX and INT return double-precision numbers (eight bytes of memory) that have a zero decimal part. One practical consequence of this difference is that the operand and value of FIX and INT are not limited to the range of integer values. For example, FIX(123,000.44) returns the value 123,000 without an error message, whereas the same operand used with the CINT function would result in an “OVerflow” (OV) error message.

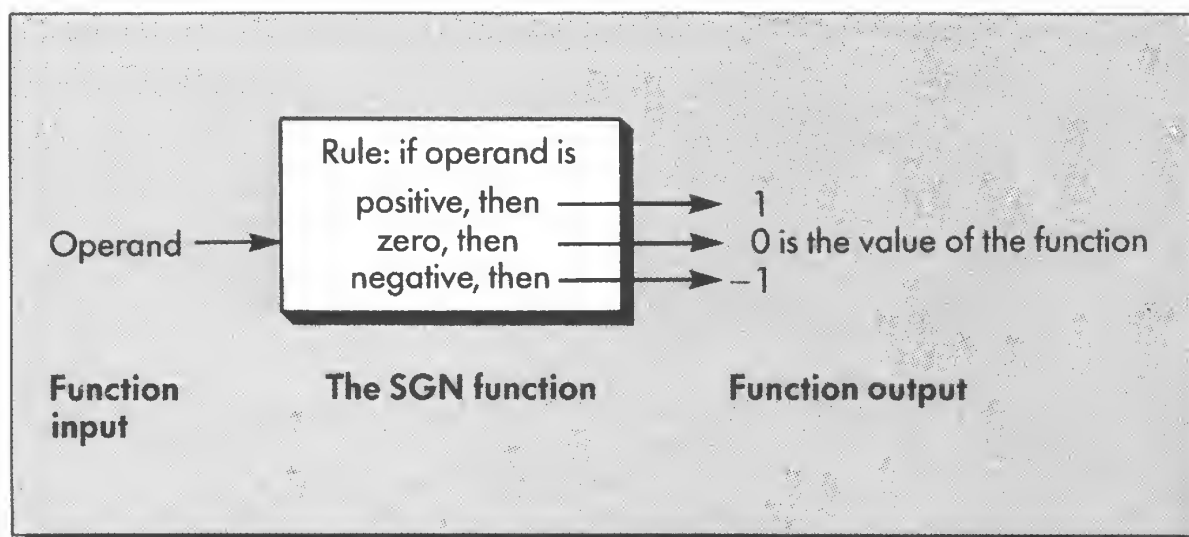


Figure 14-2. The SGN function

The difference between FIX and INT is evident only with *negative* operands. FIX truncates all digits after the decimal point. For example, enter following direct command:

```
PRINT FIX(-4.6)
-4
OK
```

FIX drops the 6 after the decimal point to return the value -4 . Contrast this result with the value of INT(-4.6):

```
PRINT INT(-4.6)
-5
OK
```

INT returns the value -5 instead of the -4 returned by FIX. For negative operands, INT returns the largest number that is equal to or less than (more negative) its operand.

The CINT, FIX, and INT Functions

All three functions take a number N and return values as shown in Figure 14-3. CINT truncates the decimal part of the operand to return a true integer value, which must be in the range of 32,768 to $-32,767$. FIX and INT return double-precision values with a decimal part equal to zero. Whereas FIX drops all digits after the decimal point, INT returns the largest number that is less than or equal to its operand.

An Example Using FIX, INT, and ABS

All right, you now know what these functions do, but what purpose do they serve? Consider the problem of converting the time in decimal hours to hours and minutes. The following program makes this conversion:

```
100 REM--NAME:HRSMIN,BA-----
110 REM--prgm converts hour value as
120 REM--decimal number to hrs, & min.
130 REM
140 CLS
```



```

150 INPUT "Time in decimal hours", TD
160 HRS = FIX(TD)
165 DM = ABS(TD - HRS)*60      ← Decimal minutes
170 MIN = INT(DM + .5)        ← Rounds off DM to nearest minute
180 PRINT "The time is " HRS "hours";
190 PRINT " and" MIN "minutes"
190 END
RUN
Time in decimal hours? 3.253
The time is  3 hours and 15 minutes
OK
RUN
Time in decimal hours? -3.253
The time is -3 hours and 15 minutes
OK

```

We've RUN this program twice to demonstrate that even a negative decimal hour will give a sensible output. For positive times, we could have used INT in place of FIX in line 160; but for a negative time (as in the second RUN), we would have gotten the hours wrong. Line 165 finds the minutes as a decimal number. ABS makes sure that the minutes always come out positive. Line 170 rounds off the value of DM to the nearest minute by first adding .5 and then truncating the decimal part of DM. Note that this program would also work if we replaced the functions FIX and INT by CINT. You to extend this program to include a calculation of seconds as well.

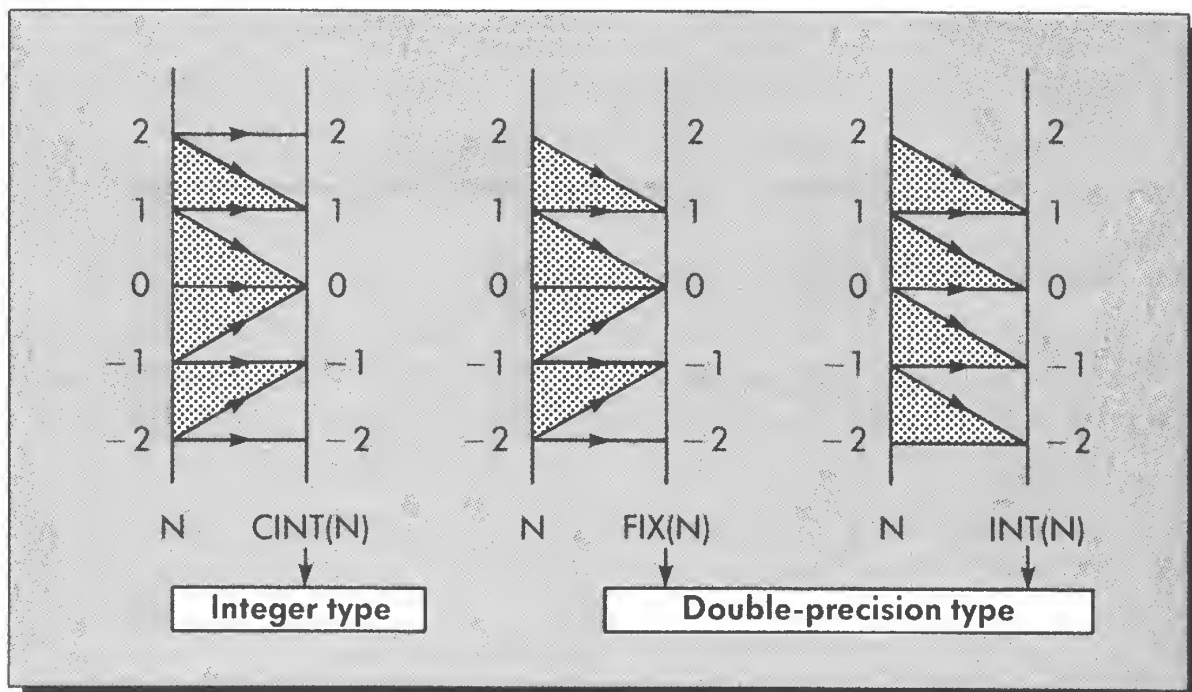


Figure 14-3. The CINT, FIX and INT functions

Trigonometry on the Model 100

We'll look now at a really interesting group of functions — the *trigonometric functions*. They are extremely useful for dealing with triangles and angles. The *tangent function*, for example, provides a tool with which to answer the following type of problem: "Suppose you are out on a safari and see a giraffe. If the giraffe is standing at a known distance away from you (say, at a riverbank) and you've measured the visual angle between the giraffe's feet and its head (with the protractor in your pocket), how tall is the giraffe?" We'll come back to this problem after we've dealt with some basics.

The three basic trigonometric functions are the *tangent*, *sine*, and *cosine*, the names of which in BASIC are TAN, SIN, and COS. We'll assume here that you're already familiar with sine, cosine, and tangent functions, but as a review, you can refer to Figure 14-4, which gives their basic definitions and BASIC names.

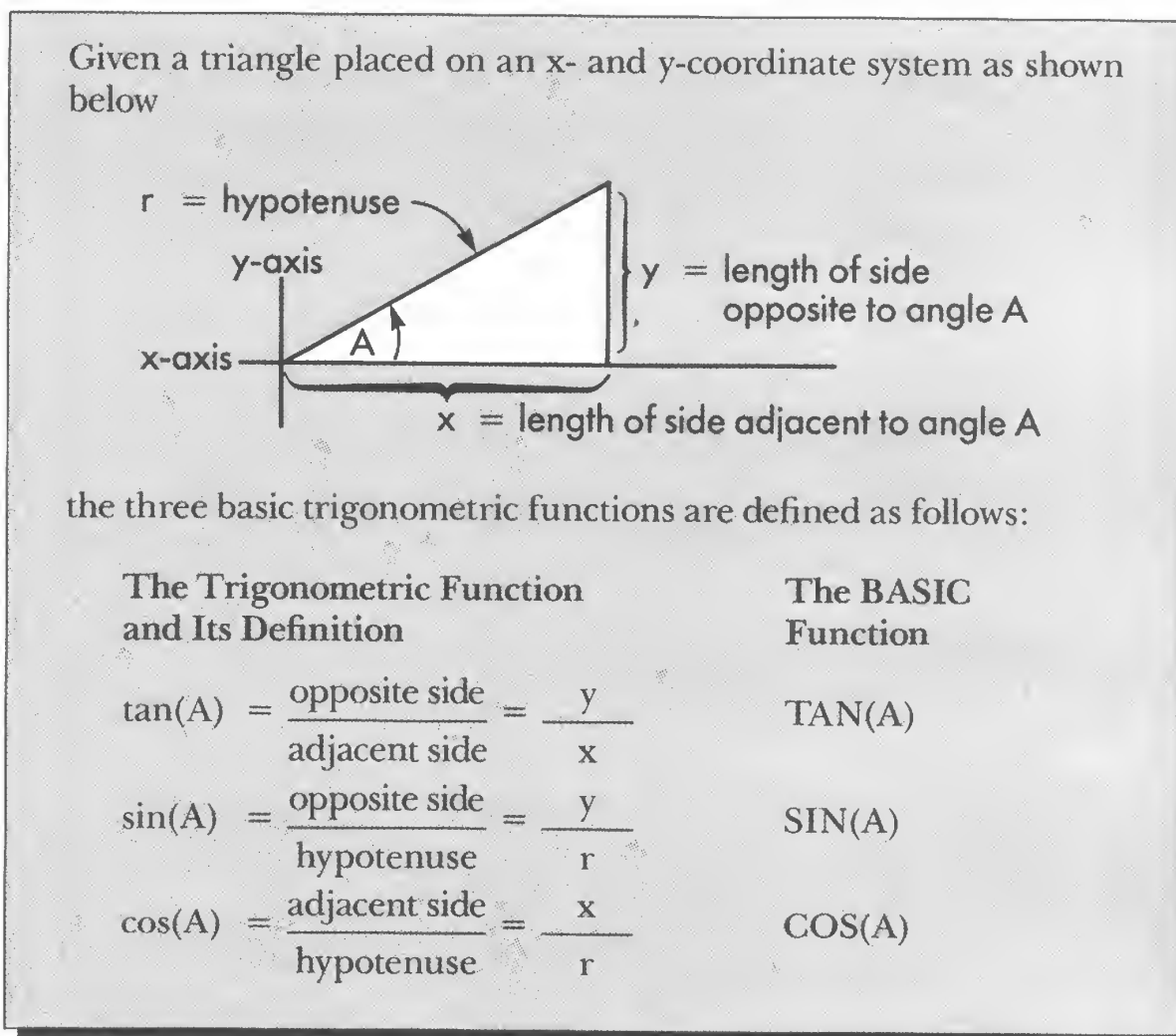


Figure 14-4: Definitions of SIN, COS, and TAN

Let's find the tangent of an arbitrary operand, say, 1.2, by using the following direct command:

```
PRINT TAN(1.2)
2.5721516221265
OK
```

The *operand* of the TAN function (the number 1.2 in our example) is the *angle* whose tangent we want to find. Most of us are used to dealing with angles in *degrees*; for example, there are 360 degrees in a circle and 90 degrees in a right angle. However, the Model 100 uses the *radian* measure (a common practice in some branches of mathematics). It's a simple matter to convert between radians and degrees: we need only multiply by a constant. (Figure 14-5 summarizes how to make this conversion.) To find the number of degrees in 1.2 radians, multiply the radian angle (1.2 in our example) by the conversion factor 57.295779513081 (which is $180/\pi$) to give the value 68.754935415698. Realize, of course, that although the Model 100 calculates to a precision of fourteen digits, most applications require far fewer digits.

The *value* of the TAN function (2.5721516221265) is the ratio of the opposite to the adjacent side, as defined in Figure 14-4. The words *opposite* and *adjacent* are defined in relationship to the operand angle.

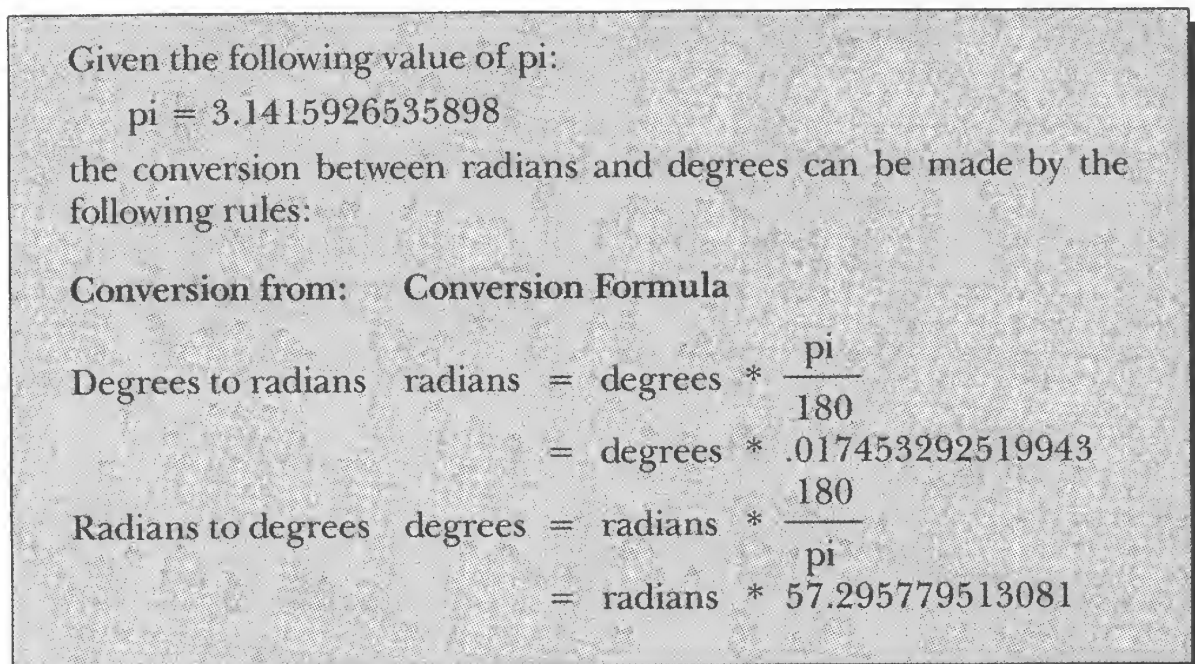


Figure 14-5: Converting between radians and degrees

Armed with this information about degrees and radians, let's write a program that asks the user for an angle in degrees and then prints out the values of the tangent, sine, and cosine of this angle.

```

100 REM--NAME:TRGFNS,BA-----
110 REM--Prgrm requests input in
120 REM--degrees and finds TAN, SIN
130 REM--and COS
140 REM
150 CLS
170 INPUT "Angle in degrees"; DEG
180 RAD = DEG*57.295779513081
190 PRINT "Tan of " DEG "degrees = " TAN(RAD)
200 PRINT "Sin of " DEG "degrees = " SIN(RAD)
210 PRINT "Cos of " DEG "degrees = " COS(RAD)
220 END
RUN
Angle in degrees? -30
Tan of -30 degrees = -.57735026918966
Sin of -30 degrees = -.5000000000000002
Cos of -30 degrees = .86602540378442
OK

```

If you've ever looked in mathematical handbooks or used a slide rule to find these functions (your only choice about fifteen years ago), you'll be impressed by the speed and accuracy of your Model 100. Notice that there is no problem with entering negative angles: a negative angle is an angle measured in the *clockwise* direction from the positive x-axis; positive angle is measured in the *counterclockwise* direction. Angles larger than 360 degrees (or 2π radians) are also acceptable.



Figure 14-6. Observing a giraffe with a protractor

Some Examples Using SIN, COS, and TAN

Now let's return to the problem posed earlier in this section: finding the height of a giraffe if we know its distance and the visual angle between its feet and its head. Referring to Figure 14-6, notice the triangle that has your eye as one of its angles, with the giraffe as the opposite side. Notice that only the opposite and the adjacent side to the angle are involved — hence, the tangent function should do the trick. From Figure 14-6, you can see that

$$\text{since } \tan(\text{DEG}) = \frac{\text{opposite side}}{\text{adjacent side}} = \frac{\text{height of giraffe}}{\text{distance to giraffe}}$$

$$\text{then height of giraffe} = \text{distance of giraffe} * \tan(\text{DEG})$$

Translating the last statement into BASIC, we arrive at this program:

```
100 REM--NAME:GIRAFF.BA-----
110 REM--Prgm finds height of giraffe
120 REM--given angle in deg. & distance
130 CLS
140 DEF SNG A-Z 'use only single prec,
150 INPUT "Distance of giraffe in feet"; DIST
160 INPUT "Angle of giraffe in degrees"; DEG
170 RAD = DEG*.01745
180 HEIGHT = CINT(DIST*TAN(RAD) + .5) ← Adding .5 causes rounding off
190 PRINT "The giraffe is" HEIGHT "feet tall"
200 END
RUN
Distance of giraffe in feet? 120
Angle of giraffe in degrees? 12.5
The giraffe is 26 feet tall
OK
```

Giraffes are very tall! Because we weren't interested in an answer more accurate than the nearest foot, we used the CINT function to return an integer value for HEIGHT. For the same reason, we also used only four digits in the degree-to-radian conversion factor in line 170.

Trigonometric functions can be used very effectively to produce interesting graphics. Let's use the SIN and COS functions to draw a circle:

```
100 REM--NAME:CIRCLE.BA-----
110 REM--Prgm draws a circle of radius
120 REM--R & places dot at center
130 REM
140 CLS
```

```

150 R = 30
160 XC = 120: YC = 32
170 '
180 PSET (XC,YC)
190 PSET (XC+R,YC)
200 FOR J = 1 TO 64 STEP 2
210   RAD = J/10
220   X = XC + R*COS(RAD)
230   Y = YC + R*SIN(RAD)
240   LINE -(X,Y)
250 NEXT
260 END

```

← Radius of circle
 ← x- and y-coord. of center
 ← Dot at center of circle
 ← Start of circle
 ← Draws circle
 ← Angle in radians
 ← x-coordinate of point on circle
 ← y-coordinate of point on circle
 ← Draws line from old to new coord.

Lines 150 and 160 determine the radius (R) of the circle and the position of its center (XC and YC), so it's an easy matter to move the circle around, change its radius, or use this program as a building block for a complex pattern of circles. The heart of the program is contained in lines 220 and 230, which determine the x- and y-coordinates of the points along the circle. The diagram in Figure 14-7 helps explain the origin of these statements.

Our circle program also includes a variation of the LINE statement we haven't previously explained. Notice that LINE in line 240 contains only one set of coordinates, with a hyphen before it; that is, the first set of coordinates specifying the first end point of the line is missing. In effect, this LINE statement draws a line from the last point drawn by a *previously*

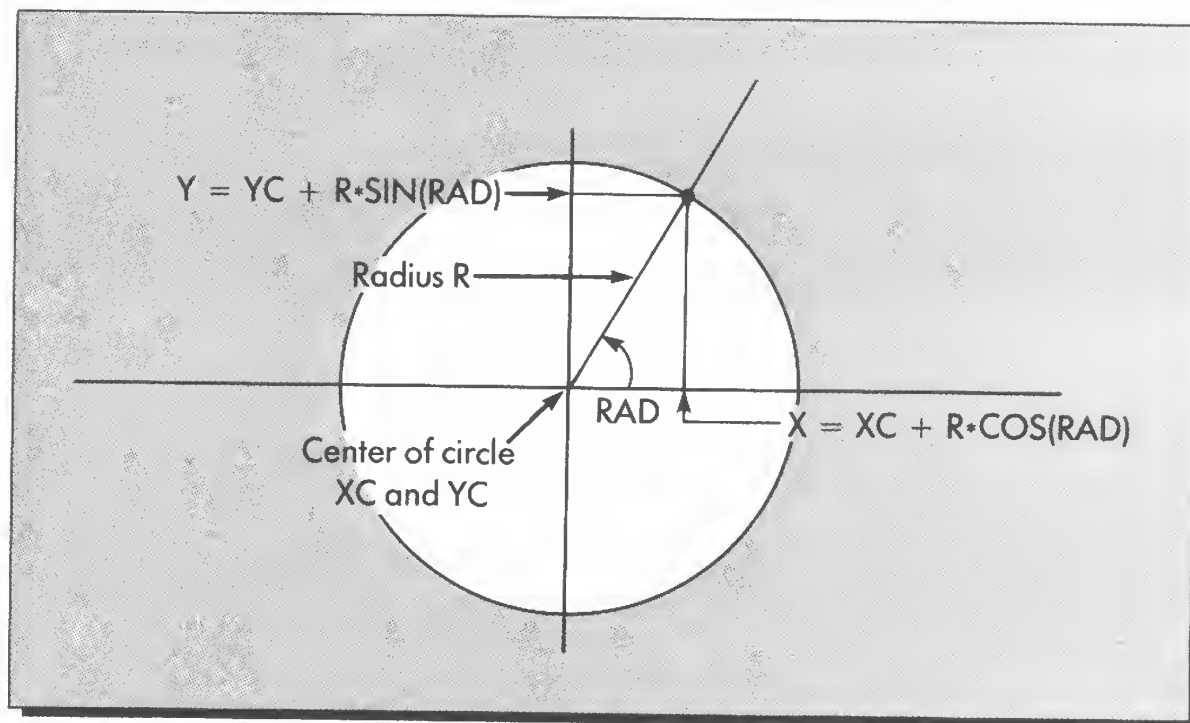


Figure 14-7: Finding the coordinates of a point on a circle

executed LINE or PSET statement. This use of LINE is convenient for drawing a series of lines joined end to end, because we need only specify the last end point of the line. The PSET statement in line 190 serves as the starting point in drawing the circle.

The Inverse Function ATN

We've described how we can use the Model 100 to find the length of a given side of a right triangle if we know one other side and one of the angles, but we often need to find an *angle*, given the lengths of two sides of a right triangle. The basic function that allows us to do this job is the inverse tangent function, ATN, an abbreviation for "ArcTanGent".

The inverse tangent function performs the opposite, or "inverse", procedure of the tangent function. The ATN function returns the angle (in radians) whose tangent is given by its operand. For example, enter the following direct command:

```
PRINT ATN(1)
,78539816329829      ← Angle in radians whose tangent equals 1
OK
```

This tells us that the inverse tangent of 1 equals the angle whose tangent is the operand (the number 1, in this example).

Frequently, we'll want the angle returned by ATN to be in degrees rather than in radians. That's easy; we simply multiply the radian angle by the number 180/pi, or 57.295779513081 (see Figure 14-5), to obtain degrees. The following program calculates the inverse tangent in both radians and degrees of an operand value entered by the user:

```
100 REM--Pgrm finds inverse tangent
110 REM
120 INPUT "Value of tangent", TN
130 RAD = ATN(TN)
140 DEG = 57.295779513081*RAD
150 PRINT "Angle in radians = ";RAD
160 PRINT "Angle in degrees = ";DEG
170 END
RUN
Value of tangent? 1
Angle in radians = ,78539816329829
Angle in degrees = 44.999999994318
OK
```

The second value in the output tells us that the inverse tangent of 1 is very nearly 45 degrees; the difference between the number shown and 45.000000000000 represents an extremely small error in the calculation — an error that has virtually no practical significance.

You may be expecting us to show you the inverse functions of the sine and cosine. Of course, these exist — they return the angle whose sine or cosine equals the operand:

Inverse sine = $\arcsin(x)$ = angle whose sine equals x

Inverse cosine = $\arccos(x)$ = angle whose cosine equals x

These functions are not part of BASIC's repertoire, however, although you can easily "construct" them yourself using the formulas given in the appendix called "Derived Functions" in your *TRS-80® Model 100 Portable Computer* manual.

BASIC Trigonometric Functions

BASIC on the Model 100 implements the four trigonometric functions shown here:

Function	BASIC Function
$\tan(\text{angle})$	TAN (angle in radians)
$\sin(\text{angle})$	SIN (angle in radians)
$\cos(\text{angle})$	COS (angle in radians)
$\arctan(\text{value})$	ATN (value) returns value in radians

Notice that all angles in BASIC are interpreted as radians. The conversion factors between degrees and radians are

Radians = degrees * $\pi/180$ = degrees * .017453292519943

Degrees = radians * $180/\pi$ = radians * 57.295779513081

Of Powers, Roots, and Logs

Mathematical calculations frequently require a number to be raised to a given power. For example, to find the area of a circle, we need to multiply π (approximately 3.14) times the radius raised to the second power (or the square of the radius, which is equivalent to multiplying the radius by itself).

We can write a BASIC statement for the area of a circle having a radius of 5 in two different ways:

```
AREA = 3.14 * 5 * 5
```

or

```
AREA = 3.14 * 5^2
```

where 5^2 means “raise 5 to the second power”, or, equivalently, find the product of $5 * 5$. The number 2 is also called the *exponent* of 5. Table 14-1 shows examples of raising 5 to different powers using standard mathematical notation and the equivalent BASIC expressions.

So much for the basics. The Model 100 is very adept at exponentiation or “raising a number to a given power”. Try the following program:

```
10 FOR EX = -3 TO 2
20   Y = 2^EX
30   PRINT EX, Y
40 NEXT
RUN
-3           ,125
-2           ,25
-1           ,5
0            1
1            2
2            4
OK
```

Perhaps you'll find these results interesting. A negative exponent means the same thing as the *reciprocal* ($1/\text{exponent}$) of a number (in this case 2) raised to the positive exponent. Also, 2 raised to the zeroth power equals 1 — a result not at all obvious!

Mathematical Notation	BASIC Notation	Result
5^0	5^0	1
5^1	5^1	$5 = 5$
5^2	5^2	$5 * 5 = 25$
5^3	5^3	$5 * 5 * 5 = 125$
5^4	5^4	$5 * 5 * 5 * 5 = 625$

Table 14-1: Examples of exponentiation

Taking the Square Root with SQR

Exponents needn't be integers; they can be fractions or decimals. A very common fractional exponent is .5 or 1/2; it has the same effect as taking the *square root* of a number. In fact, taking square roots is such an important and frequently used procedure that BASIC has a special function for just this purpose, called SQR. The following is an example of how SQR can be used:

```
PRINT SQR(25)
5
OK
```

Well, you already knew that the square root of 25 is 5 (because $5 * 5$ equals 25, right?), so let's try a less obvious example. Let's use the SQR function to solve the practical problem of finding the length of the hypotenuse of a right triangle, given the lengths of the other two sides:

```
10 INPUT "Base of triangle"; BASE
20 INPUT "Height of triangle"; HEIGHT
30 HYPOTENUSE = SQR(BASE^2 + HEIGHT^2)
40 PRINT "The hypotenuse =" HYPOTENUSE
50 END
RUN
Base of triangle? 3
Height of triangle? 4
The hypotenuse = 5
OK
```

This result tells us that the diagonal distance across a rectangle of dimension 3 by 4 equals 5. Notice that we also had the opportunity to use exponentiation in line 30.

The Number e, EXP and LOG

There are some very special numbers in our universe, such as pi, that crop up frequently whenever circles are involved in a problem. Another such number is called e; its value is approximately 2.718283. It looks perfectly ordinary, but anyone who's taken a course in calculus knows that it appears quite "naturally" in many types of rather sophisticated calculations. Though we can't explain its meaning here, we can show you the two BASIC functions related to it.

Power of e with the EXP Function

The EXP function raises the number e to the power indicated by the operand. For example,

```
PRINT EXP(2)
7.38905609893      ← e raised to the second power (equal to e2)
OK
```

tells us that the number e raised to the second power equals 7.38905609893. The following program further explores this function:

```
10 INPUT "Exponent of e"; X
20 Y = EXP(X)
30 PRINT "e raised to the power " X "equals" Y
40 END
RUN
Exponent of e? 1
e raised to the power 1 equals 2.7182818284588
RUN
Exponent of e? -1
e raised to the power -1 equals .36787944117145
OK
```

The first RUN returns the value of e because e^1 equals e; the second RUN gives the value of e raised to the -1 power, which, by definition, equals $1/e^1$, or $1/e$.

Taking the Natural Logarithm with LOG

The LOG function is the inverse of the EXP function: whereas EXP(X) raises e to the power X, LOG(Y) finds the exponent to which e must be raised in order to give the number Y. The standard mathematical expression for BASIC's LOG(Y) is $\ln(y)$. This form of the logarithm is the "natural" logarithm in contrast to the logarithm to the base 10. The relationship between raising e to a given power and the logarithm of a number is summarized in Figure 14-8.

In mathematical notation,
if $y = e^x$ then $x = \ln(y)$
or in BASIC's notation,
if $Y = \text{EXP}(X)$ then $X = \text{LOG}(Y)$

Figure 14-8: Relationship between exponentiation and the natural logarithm

You can familiarize yourself with the LOG function with the help of this program:

```
10 INPUT "enter a number"; X
20 Y = LOG(X)
30 PRINT "the natural logarithm = " Y
40 END
RUN
enter a number? 1
the natural logarithm = 0
OK
RUN
enter a number? 2.7182818284588      ← Equal to e
the natural logarithm = .9999999999986
OK
```

The result of the second RUN reflects the fact that e^1 equals e .

Exponentiation and the SQR, EXP, and LOG Functions

Shown below are four mathematical expressions and their implementation in BASIC on the Model 100:

Mathematical Expression	BASIC Expression
$y = x^3$	$Y = X^3$
$y = x$	$Y = \text{SQR}(X)$
$y = e^{4.5}$	$Y = \text{EXP}(4.5)$
$y = \ln(x)$	$Y = \text{LOG}(X)$

Random Numbers with RND

Most of the time we use our computer to produce very precise, predictable results. Computers excel at following instructions to the “bit”; they are not subject to the vagaries (well, not usually, anyway!) so characteristic of humankind. However, some computer applications require that the computer do the thing it “likes” least — produce random numbers. *Random numbers* have no patterns in the way their digits appear. Applications range from “playing” games of chance, like poker or similar games involving dice, to simulating the behavior of gas molecules to graphic displays having some degree of randomness.

BASIC makes it easy for us to generate random numbers. As a first experiment, enter the following direct command:

```
PRINT RND(1).  
.59521943994623  
OK
```

The BASIC word *RND* stands for “RaNDom”. The output of RND is a random fourteen-digit number having a value between 0 and 1. What is the meaning of the operand, the number 1 in parentheses? The following program reveals the operand’s purpose:

```
10 FOR J = 3 TO 0 STEP -1  
20   PRINT RND(J)  
30 NEXT  
RUN  
.59521943994623  
.10658628050158  
.76597651772823  
.76597651772823  
OK
```

The first three numbers in the output for values of J equal to 3, 2, and 1 are different random numbers. The last number in the output, however, repeats the previous random number, because the last number is generated with a value of J equal to 0. The general rule is this: if the operand of RND is 0, RND will repeat the previous random number; if the operand is positive, a new number will be generated with each execution of RND.

Now run this program again. The *same* random numbers appear in the output! Your Model 100 produces the *same* sequence of random numbers every time a program containing a sequence of RND statements is run. To differentiate a true random number generator, which would never (well, almost never) produce the same output twice, the type of random number generator inside the Model 100 is a *pseudo-random number generator*.

To show how we can use the computer’s capability to generate a sequence of random numbers, let’s write a program that “rolls” a die. Since every roll results in a random integer number from 1 to 6, we must translate random decimal numbers between 0 and 1 to random integers from 1 to 6.

The following program "throws the die" ten times:

```
10 REM--NAME:DICE,BA-----
20 CLS
30 FOR J = 1 TO 10
40   NDICE = CINT(RND(1)*6) + 1
50   PRINT NDICE;
60 NEXT
OK
RUN
 4  1  5  4  5  2  3  6  4  3
OK
```

CINT(RND(1)*6) in line 40 produces integer values between 0 and 5 so we need only add a 1 to get numbers ranging from 1 to 6. Note that we used the number 1 as the operand of RND, though any positive number or expression resulting in a positive number would have had the same effect.

If you run this program again, you'll get the same sequence of dice throws. Such repetition is not objectionable in most applications, but if your computer is rolling the dice for a poker game, some people might get rather upset! Fortunately, there is a partial solution to this problem: instead of printing out the first number generated by RND, you can start printing the random numbers generated *later* in the sequence. (See your *TRS-80® Model 100 Portable Computer* manual under "RND" for details.)

We can also use the RND function to effectively create various graphics effects; for example, the following program draws a series of lines that suggest an explosion:

```
10 REM--NAME:EXPLOS,BA-----
20 CLS
30 XC = 120   'x-center of explosion
40 YC = 30    'y-center of explosion
50 FOR J = 1 TO 40
60   X = XC + (RND(1)-.5)*160
70   Y = YC + (RND(1)-.5)*50
80   LINE (XC,YC) - (X,Y)
90 NEXT
```

The LINE statement in line 80 draws lines that always start at the coordinates XC and YC, the center of the "explosion". Lines 60 and 70 define the random endpoint of each line drawn. The possibilities for creating interesting patterns with RND, PSET, and LINE are endless!

The RND Function

The following statement returns a fourteen-digit pseudo-random number:

```
40 PRINT RND(N)
```

If N is positive, this statement will print out a *different* random number each time it is executed. If N is zero, the *same* random number will be printed out with each execution of the statement.

Summary

Numerical functions are a powerful addition to your repertoire of BASIC commands, useful not only in scientific applications but also in business and graphics applications. Though you may not fully understand the mathematical concepts underlying some of these functions, you can experiment and learn more about them.

In this chapter we have introduced four groups of numeric functions. The first includes the ABS function which removes the sign of a number, and the SGN function, which returns a + 1 for a positive number and a - 1 for a negative number. The second group, made up of the functions CINT, FIX, and INT, all return values without decimal parts; CINT returns a value of the integer type. The third group contains the four trigonometric functions SIN, COS, TAN, and ATN. These functions are invaluable when dealing with angles, and we can also use them to create various graphic shapes. The fourth group is composed of the SQR, EXP, and LOG functions, all of which are related to exponents.

In addition to these groups of functions, we also explained the use of the RND function, which generates random numbers. This function is useful in games, in a variety scientific simulations, and in graphics displays.

Exercises

1. Write a program that draws a “cycloid”, which is a curve traced out by a point on a *rolling* wheel; this curve can be generated on a computer by drawing a circle while moving the center of the circle.
2. Use the SIN function to create a wave pattern on your screen.
3. Use the RND function to create a pattern of random dots within a box drawn by LINE.

Solutions

1. This program draws a cycloid:

```
100 REM--NAME:CYCLD,BA-----
110 REM
120 CLS
130 R = 20 'radius of moving circle
135 SP = 1 'speed of moving circle
140 JLIMIT% = (238 - 2*R)/SP
145 '
150 FOR J% = 0 TO JLIMIT%
160   RAD = J%/6
170   X = R*COS(RAD) + J%*SP + R
180   Y = R*SIN(RAD) + 32
190   PSET (X,Y)
200 NEXT
210 END
```

By changing the variable SP in line 135, you can create various interesting graphics effects.

2. This program draws five sine “waves” to produce a wave pattern:

```
100 REM--NAME:WAVE,BA-----
110 REM
120 CLS
130 PHASE = 1
140 '
150 FOR WAVE% = 1 TO 5
160   Y1 = 20 + WAVE%*4
170   FOR X% = 40 TO 200
180     Y = Y1 + 10*SIN(X%/5)
185     X = X% + WAVE%*PHASE
190     PSET (X,Y)
200   NEXT X%
210 NEXT WAVE%
220 END
```

← Shift in x between sine waves

← Draws five curves

← Initial y-coordinate of each wave

← Draws one sine wave

By changing the value of PHASE in line 130, you can, in effect, get a different perspective on the set of waves.

3. The following program draws 300 random dots within a box:

```
100 REM--NAME:RNDOTS,BA-----
110 REM
120 CLS
130 LINE (40,10) - (200,50),1,B 'box
140 '
150 FOR J% = 1 TO 300          '300 dots
160   X = RND(1)*160 + 40      'x-scaling
170   Y = RND(1)*40 + 10      'y-scaling
180   PSET (X,Y)
190 NEXT
200 END
```

String Functions

Concepts

- Changing strings to numbers and vice versa
- String generation
- String length
- Keyboard input without
- String search
- String manipulations

Instructions

SPACE\$, VAL, STR\$, STRING\$, LEN, INPUT\$, INKEY\$, LEFT\$, RIGHT\$, MID\$, INSTR

We've seen many interesting and powerful ways that numeric functions can be used to manipulate numbers. In this chapter we'll explore the use of string functions, which manipulate strings. Remember that a *string value* is any sequence of characters, including numbers, bracketed by quotation marks. Though we can't take the square root or sine of a string, BASIC allows us to manipulate and transform strings in many different and interesting ways.

In general, functions are rules by which an input determines an output. *Numeric functions* involve only numbers or numeric expressions, whereas *string functions* usually involve at least one string or string-variable or expression in the output or the input. Most string functions also include one or more numeric variables. Let's begin with one of the simplest string functions, SPACE\$.

The SPACE\$ Function

The SPACE\$ function returns a string consisting of nothing but spaces, the number of which is determined by the input, or operand (the number in parentheses following STRING\$), as shown in the following example:

```
10 SPACEOUT$ = "spaced" + SPACE$(5) + "out"
20 PRINT SPACEOUT$
30 END
RUN
spaced      out          ← Five spaces between 'spaced' and 'out'
OK
```

The function SPACE\$(5) returns a string value consisting of five spaces. As shown in the above example, this string of spaces can be “concatenated”, or added, to other strings.

SPACE\$ is useful when we need to insert a large number of spaces or if spaces are repeatedly required, as in the following example:

```
10 X$ = "+" 'initial value of X$
20 FOR J = 1 TO 12
30     X$ = X$ + SPACE$(2) + "+"
40 NEXT
50 PRINT X$
60 END
RUN
+ + + + + + + + + + + +
OK
```

Such a pattern of characters might be useful as markers or headings. Notice the “summing” procedure used to construct the string X\$ by means of the FOR...NEXT loop — we’ve used it before to find the totals of numbers. The operand of the SPACE\$ function (the number 2 in this example) can be a variable or a numeric expression. You might try the following: replace the number 2 here with the variable J — and see what happens!

The SPACE\$ Function

The SPACE\$ function returns a string consisting of spaces, the number of which is specified by the operand, as illustrated below:

X\$ = SPACE\$(5)

Operand determines number of spaces

Returns string function consisting of five spaces

Operands can be numbers, variables, or numeric expressions having values in the range 0-255.

The VAL AND STR\$ Functions

As you know, BASIC makes an important distinction between *number* and *string* values and variables. Numbers are composed of digits; a number always has one space after it, as well as a space reserved for its sign, and numbers can, of course, be mathematically manipulated. Strings, on the other hand, are composed of any character or sequence of characters, including numbers and special symbols such as graphics symbols.

In some programming applications, we must convert a string made up of digits to a true number that can be treated mathematically. The converse is sometimes also required; that is, we must convert a number to a string value having the same digits. The two BASIC instructions that perform this function are the VAL and STR\$ functions.

Converting Strings to Numbers with VAL

The function VAL converts a string to a numeric value. Consider the following example:

```
10 M$ = "50000 dollars"
20 PRINT M$
30 PRINT VAL(M$)
40 END
RUN
50000 dollars
50000
OK
```

The first value in the output is a string value of M\$, and the second value is the *numeric* value VAL(M\$), which accounts for the space reserved for the sign of the number in front of the 50000. The value returned by VAL is the numeric equivalent of its operand. Note that VAL ignored the word “dollars” — it simply ignores all nonnumeric characters. However, if the first character in the string is nonnumeric, VAL returns a 0, as in this example:

```
PRINT VAL("$50000")
0
OK
```

A fast way to lose \$50,000!

Converting digits within a string to a numeric value is necessary whenever we must manipulate these digits mathematically. Consider, for example, the following program, which converts time within a string in AM and PM notation to twenty-four hour time:

```
10 X$ = "6 PM is Fido's dinnertime"
20 T = VAL(X$) + 12
30 PRINT T "hrs. is Fido's dinnertime"
40 END
RUN
18 hrs. is Fido's dinnertime
OK
```

VAL(X\$) in line 20 returns the number 6, which is then added to 12 to give 18.

Changing Numbers to Strings with STR\$

The STR\$ function complements the VAL function: whereas VAL converts a string to a numeric value, STR\$ converts a numeric to a string value. For example:

```
PRINT STR$(18) + "hrs"
18hrs
OK
```

STR\$(18) returns the string “18”, which is concatenated to the string “hrs”. Notice that the space preceding the number (always reserved for the number’s sign) is still present in the string; however, the space *after* the number is absent in the string — that’s why there’s no space between “18” and “hrs” in the output.

The VAL and STR\$ Functions

VAL converts the leading digits of a string value to a numeric value, whereas STR\$ converts a numeric to a string value.

VAL("123")	← Returns the <i>numeric</i> value 123
STR\$(123)	← Returns the <i>string</i> value '123'

Making Strings with STRING\$

Some programming applications require that we use strings having many repetitions of a particular character. For example, a dashed, dotted, or solid line is made up of repetitions of the characters "-", ".", and ASCII character 241, respectively. The STRING\$ function makes it easy to construct strings that consist of such repetitions of a single character. There are two variations of this function, which we'll explore in the following two sections.

Using Keyboard Characters in STRING\$

In the first variation of STRING\$, the character to be used as the building block for the string is specified as a string character in the usual manner. Consider the following example:

```
10 X$ = STRING$(40,"$")
20 PRINT X$;
30 END
RUN
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
OK
```

The function STRING\$(40,"\$") returns a string of forty dollar signs. The first operand specifies the number of repetitions; it can be any numeric expression that returns a value from 1 to 255. The second operand must be a string value; it specifies the character that is to make up the value of STRING\$.

Using ASCII Code Values in STRING\$

The second variation constructs a string of identical characters in the same manner as the first variation, except that the character to be used in STRING\$ is specified by its *ASCII code value*. The following example has the same output as our previous one:

```
10 X$ = STRING$(40,36)
20 PRINT X$;
OK
RUN
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
OK
```

As before, the first operand specifies the number of repetitions, and the second specifies the character to be repeated in the value of STRING\$. However, the character is identified by its ASCII code value, 36 in this example. We can use any of the printable ASCII character codes in this manner. In the following example, we use a selection of graphics characters having ASCII code values ranging from 229 to 255 to create various patterns:

```
100 REM--NAME:STRPAT.BA-----
110 REM--creates patterns using STRING$
120 REM--fn. and graphics characters
130 CLS
140 INPUT "Step (1-5)"; ST
150 CLS
160 STRT = 229 'first ASCII value
170 FOR J = STRT TO STRT+ST*5 STEP ST
180   PRINT STRING$(40,J);
190 NEXT
200 END
```

Each different value entered results in a different pattern. You may enjoy trying other variations, such as changing the value of the first ASCII code value in line 160. Notice that the STRING\$ function in line 180 uses the numeric *variable* J to specify the particular character to be printed as a chain of characters.

The STRING\$ Function

The STRING\$ function returns a string consisting of repetitions of a particular character. Its two operands specify the number of repetitions and the character to be repeated. There are two variations of this function. An example of the first is the following:

STRING\$(20,"\$") ← Returns string of twenty dollar signs

↑ ↑
Specifies character in value of STRING\$
Specifies number of repetitions of specified character

The second variation uses the ASCII code value of the character in place of the character itself. The following is identical in effect to the previous example:

STRING\$(20,36) ← Returns string of twenty dollar signs

Input Using INPUT\$ and INKEY\$

By now you are well acquainted with the INPUT statement that we introduced in Chapter 7. For most purposes, it's the most convenient tool for allowing a user to input information into a running program. However, two other methods of entering user information may be more appropriate for certain applications. These are the instructions INPUT\$ and INKEY\$. The principal difference between INPUT and these two new functions is that INPUT requires the user to press **ENTER** to signal the computer to "read" the information typed in, whereas INPUT\$ and INKEY\$ do not. (Incidentally, we're intentionally not being too picky about identifying INPUT\$ and INKEY\$ as functions; there are also good reasons to think of them as string variables.)

The INPUT\$ Statement

Let's look first at the easier of the two functions — INPUT\$. The following example shows how we can use INPUT\$ to enter keyboard characters without pressing **ENTER**:

```
10 PRINT "Press any two keys:"
20 RSP$ = INPUT$(2)
30 PRINT "your input was " RSP$
40 END
```

When this program is RUN, the following appears on the screen:

```
RUN
Press any two keys:
```

That's all that happens; there's no question mark, not even a blinking cursor. The program is simply waiting for the user to press two keys. All right, let's press the keys *h* and *i*. The instant we've pressed *i*, the program responds with the following:

```
your input was hi
OK
```

Notice that we didn't have to press **ENTER** for the program to read and process our input. Also, note that as we typed *h* and *i*, the letters *h* and *i* didn't immediately appear on the screen the way they would have had we used an INPUT statement in line 20. INPUT\$ doesn't echo input information in the way that INPUT does.

Let's take a closer look at how this works. When program execution reaches the BASIC word INPUT\$(2) in line 20, it waits for the user to press *two* keys. The number in parentheses after INPUT\$ specifies the number of characters that must be keyed in before program execution resumes. This number can be anywhere between 1 and 255.

Once the required number of keys has been pressed, the value of INPUT\$(2) is assigned to the variable RSP\$. Whereas INPUT can be used by itself in a BASIC line, INPUT\$ cannot. INPUT\$ is a *function* that returns a string value from the keyboard, and as such, it must appear in conjunction with a BASIC statement that in some way processes the value of the function.

One application in which we can use INPUT\$ very effectively is a menu-selection program, such as the program "MENU1.BA" in Chapter 11. Line 230 of this program is a typical INPUT statement:

```
230 INPUT "Selection (1-4)"; SEL
```

which requires that the user press **ENTER** to complete the selection entry. If you wish to rewrite this menu program so that the user can make the selection by just pressing keys 1-4 without pressing **ENTER**, replace the above line 230 with the following three lines:

```
229 PRINT "To select, Press Key 1-4"
230 SEL$ = INPUT$(1)
231 SEL = VAL(SEL$)
```

Line 229 replaces the prompt of our original INPUT statement, line 230 assigns the keyboard entry to the string variable SEL\$, and line 231 converts the string value of SEL\$ to a numeric value that can be used in the subsequent ON...GOSUB statement.

The INPUT\$ Statement

INPUT\$ is a string function that returns a string value containing a specified number of characters from the keyboard without the requirement of pressing **ENTER**, as shown in the following example:

```
30 CODE$ = INPUT$(5)    ← The number 5, the operand, specifies the
                        number of characters to be keyed in.
```

INPUT\$(5) will return a string of five characters as soon as five keys have been pressed. The operand of INPUT\$, which specifies the number of characters to be keyed in, can be any numeric expression that returns a value in the range 1-255. All characters except **BREAK** are accepted.

INKEY\$ — An Input Function That Can't Wait

INKEY\$ is somewhat similar to INPUT\$. Both are string functions that return characters from the keyboard; neither requires **ENTER** to input information, and neither echoes the keyboard entry.

There are, however, several important differences. The following program, which is very similar to the first program in the previous section, demonstrates one difference:

```
10 PRINT "Press a Key:"
20 RSP$ = INKEY$
30 PRINT "your input was " RSP$
40 END
```

RUN this program and be ready for a surprise:

```
RUN
your input was
OK
```

Immediately after you enter RUN, your program PRINTs "your output was" and the BASIC prompt appears. The program is finished before you have had a chance to press a key (unless you're *very* fast!). The problem is that INKEY\$ *doesn't wait* for user input the way INPUT\$ does. When INKEY\$ is executed, it checks to see if a key has been pressed. If you manage to press a key before INKEY\$ is executed — very unlikely in the program above — INKEY\$ will return the string value associated with the pressed key. (You'll see that in our next example.) But if you haven't pressed a key by the time INKEY\$ is executed, instead of waiting for you to make an entry, INKEY\$ returns a *null string* — a string without any contents. That's what line 30 tells you; the value of RSP\$ is nothing.

We can change this program so that it waits for user input by inserting a new line, line 25:

```
10 PRINT "Press any Key"
20 RSP$ = INKEY$
25 IF RSP$ = "" THEN 20
30 PRINT "your input was " RSP$
40 END
RUN
Press any Key
your input was h
OK
```

← Program waits; user presses key h

This program *does* wait for user input (we pressed the h key). Notice that this entry is not echoed on your screen (as would be the case with INPUT), although it is PRINTed out by line 30.

This program waits for user input because line 25 causes program execution to loop back to line 20 as long as INKEY\$ returns a null string, that is, as long as no key has been pressed since the previous execution of INKEY\$. However, the instant a key is pressed, the value of RSP\$ will cease to equal " " — the null string — and program execution will continue with line 30. Another important difference between INKEY\$ and INPUT\$ is that INKEY\$ reads only one character from the keyboard, whereas INPUT\$ waits until a specified number of keys have been pressed.

INKEY\$ is a useful tool to produce user-initiated program interrupts. Consider the following program that simulates a stopwatch:

```

100 REM--NAME:STPPTH.BA-----
110 REM--stopwatch: begins with RUN,
120 REM--ends when any key is pressed;
130 REM--from 0 to 60 seconds only,
140 CLS
150 PRINT "Press any key to stop count"
160 T1 = VAL(RIGHT$(TIME$,2))      ← Initial time (seconds only)
170 '
180 A$ = INKEY$                    ← INKEY$ trap
190 IF A$ = " " THEN 180
200 '
210 T2 = VAL(RIGHT$(TIME$),2)      ← Final time (seconds only)
220 IF T2<T1 THEN T2 = T2 + 60      ← Ensures positive time elapsed
230 PRINT "Elapsed time =" T2 - T1  ← Prints time elapsed
240 END

```

This program works only for times shorter than sixty seconds, though it wouldn't be hard to extend its capabilities to longer elapsed times. When this program is RUN, the initial time in seconds is assigned to the variable T1 in line 160. The program is "trapped" in lines 180 and 190 as long as A\$ equals the null string, that is, as long as a key has not been pressed. When a key is pressed, program execution "drops through" to the statements that follow. Line 210 assigns the final time in seconds to T2, and line 220 ensures that if the value of TIME\$ during this counting process has passed a sixty-second mark, sixty seconds are added to the final time T2. Finally, line 230 prints out the time difference $T2 - T1$, the elapsed time. (Don't worry about the function RIGHT\$. We'll explain it in the next section.)

The INKEY\$ Function

INKEY\$ returns a single character from the keyboard. Like INPUT\$, INKEY\$ does not require the user to press **ENTER** to complete the entry, nor does it automatically display the keyboard character on the screen. But unlike INPUT\$, INKEY\$ does not wait for user input. If no key is pressed, INKEY\$ returns a null string, written as "". The following statements are an example of how INKEY\$ may be used:

```
30 VAL$ = INKEY$: IF VAL$ = "" THEN 30
```

This line can be used to await a key press in a program. The rest of the program is executed only after a key is pressed.

Manipulating Strings

In this section we'll introduce some of the more interesting and powerful string functions. We can do practically any kind of string manipulation with these functions, including word processing, games involving words and sentences, formatting text, and cryptography.

Finding the Length of a String with LEN

The function LEN returns the "LENgth" or number of characters in a string. The following example illustrates how it can be used:

```
10 WRD$ = "supercalifragilisticexpialidocious"  
20 PRINT LEN(WRD$)  
30 END  
RUN  
 34  
OK
```

The output shows that the operand of LEN, WRD\$, has thirty-four characters.

One of the many uses of this function is in automatically centering text, such as a title or phrase. We've previously centered text by counting the number of characters in the text line, subtracting that number from the desired width of the page, and dividing by two; the resulting number gives

us the number of spaces in front of the text. Now we can use the LEN function to find the text of the title for us. The following example prints out a phrase that is automatically centered without the need to count characters:

```
100 PHRASE$ = "Frankenstein bytes chips"
105 PGWID = 40
110 SP = ((PGWID - LEN(PHRASE$))/2)
120 PRINT TAB(SP) PHRASE$
130 END
```

← Page width
← Number of spaces

The variable PGWID is the width of the page, taken to be forty in this example. Line 110 calculates the number of spaces (SP) to be inserted in front of the PHRASE\$. We used TAB to insert these spaces, though we could have used SPACE\$ instead.

Extracting the Left Part of a String with LEFT\$

The LEFT\$ function is one of three BASIC functions that extract a portion of a string. As its name implies, LEFT\$ returns the left part of a string, as shown in the example below:

```
10 ACCT$ = "N234 Repair"
20 NUMBER$ = LEFT$(ACCT$,4)
30 PRINT NUMBER$
40 END
RUN
N234
OK
```

The function LEFT\$ returns the first four characters of the string ACCT\$, counting from the *left*. As you can see, this function has two operands with the following meaning:

LEFT\$(ACCT\$, 4)

Number of characters to return, counting from left

Parent string from which LEFT\$ returns the four left-most characters

If, in our above program, we replace the number 4 with 11 — the length of ACCT\$ — or any number larger than 11, the entire string “N234 Repair” would be returned.

Extracting the Right Part of a String with RIGHT\$

The function RIGHT\$ is identical to LEFT\$ except that it returns the right part of a string. To see how it works, substitute the word *RIGHT\$* for the word *LEFT\$* in line 20 of our previous program:

```
10 ACCT$ = "N234 Repair"
20 DEPT$ = RIGHT$(ACCT$, 6)
30 PRINT DEPT$
40 END
RUN
Repair
OK
```

The function RIGHT\$(ACCT\$, 6) returns the six *right-most* characters of ACCT\$. If we substitute the number 11 — equal to LEN(ACCT\$) — for the number 6, RIGHT\$ will return the whole string ACCT\$. The following summarizes the operands of RIGHT\$:

RIGHT\$(ACCT\$, 5)

Number of right-most characters RIGHT\$ returns

Parent string from which characters are extracted

Our example illustrates how RIGHT\$ can be used to extract the right portion of a string. Such “extractions” are often very useful when dealing with information such as telephone numbers, inventory, and addresses.

Our last example is for pure, unadulterated fun; as far as we can see, it has no practical application whatsoever. It does, however, use both LEFT\$ and RIGHT\$ functions.

```
100 REM--NAME:METAMO,BA-----
110 REM--"Metamorphosis", based on a
120 REM--cartoon by Louis Phillips
130 REM
140 CLS
150 X1$ = "caterpillar"
160 X2$ = "butterfly"
170 FOR J = 0 TO 11
180     L$ = LEFT$(X2$,J)
190     R$ = RIGHT$(X1$,11 - J)
200     PRINT L$ + R$
210 NEXT
220 END
```

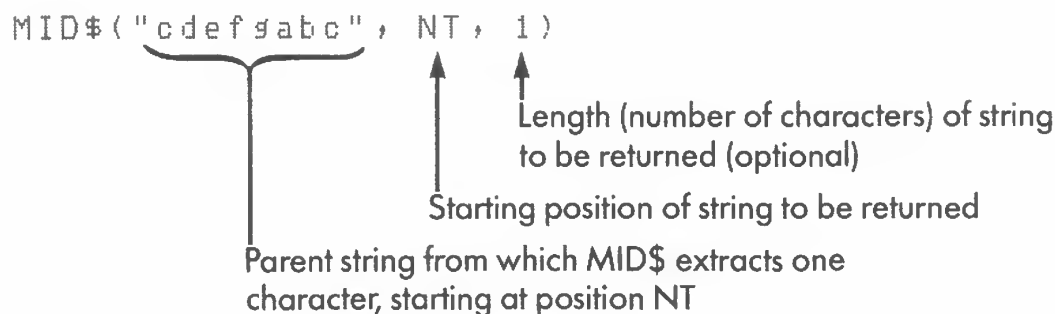
Try it — we won’t spoil the effect by showing you the output here!

Extracting Any Part of a String with MID\$

The MID\$ function is perhaps the most versatile and useful of all the string functions. It can extract *any* part of a string — in particular, the “MIDdle”. Try the following example:

```
10 INPUT "Note number (1-8)"; NN
20 NTE$ = MID$("cdefgabc",NN,1)
30 PRINT "The note is " NTE$
40 END
RUN
Note number (1-8)? 3
The note is e
```

Line 10 assigns a number — 3 here — to the variable NN. As you can see, the MID\$ function in line 20 returns the string value e, which is then printed out by line 30. Notice that MID\$ has *three* operands. The first is what we’ll call the *parent string*, from which an extraction is to be made. The second and third are numeric values that determine exactly what part of the parent string is to be returned. Their specific functions are explained in the following diagram:



In this example, the parent string is a sequence of letters defining the names of the notes of the C-major scale. The second operand, a variable in this example, defines the location of the character in the parent string to be returned by MID\$. The third operand specifies the number of characters to be returned by MID\$, which in our example is 1. This third operand is optional; if it is omitted, MID\$ will return the entire portion of the parent string, starting at the position specified by the second operand. In general, the MID\$ function can return any part of a string; therein lies its versatility and power, and also its complexity.

If you like playing with words, you'll enjoy our second example:

```
100 REM--NAME:REVERS,BA-----
110 REM--user enters phrase, prgm
120 REM--Prints out phrase backwards
130 CLS
140 INPUT "some characters"; CHAR$
150 L = LEN(CHAR$)
160 '
170 FOR J = L TO 1 STEP -1
180   PRINT MID$(CHAR$,J,1)
190 NEXT
200 END
RUN
some characters? frankenstein
nietsneknarf
OK
```

What? Read "frankenstein" backward — yes, it's the dreaded "nietsneknarf"! We can't resist at least one more RUN:

```
RUN
some characters? able was I ere I saw elba
able was I ere I saw elba
OK
```

Hmm ... a palindrome (it reads the same from either end). This one supposedly was uttered by Napoleon. Try some of your own words or phrases; you're bound to come up with some interesting surprises.

The keys to this program are line 180 and the FOR...NEXT loop, which "steps" backward from L to 1. The first letter printed by line 180 is the *last* letter of the value of CHAR\$, the word entered by the user. The next letter printed is the second letter from the left of the value of CHAR\$, and so forth. This process continues until, finally, the first letter of the entered word is printed.

Extracting letters from a string is useful whenever we need to manipulate individual letters. A variation on this theme is the problem of turning lowercase letters into uppercase letters. Try this on your own: set up a loop in which you use MID\$ to extract one letter at a time; then get its ASCII code number via ASC, subtract 32 to get the code number of the corresponding capital letter, and then turn that code value back into a letter and PRINT it.

The Position of Characters Within a String Using INSTR

The functions LEFT\$, RIGHT\$, and MID\$ give us the tools to extract any part of a string. Sometimes, however, we must also find the *position* of one or more characters within a given parent string. For example, if we wanted to use LEFT\$ to extract the last name of the string “Frankenstein, Jerry”, we’d need to first find the number of characters in “Frankenstein”, which is equivalent to the number of characters in front of the comma. The INSTR instruction is designed to perform this kind of function.

The following example shows how we can use INSTR to find the position of the comma at the end of the last name, “Frankenstein”:

```
10 NAM$ = "Frankenstein, Josephine"
15 '      12345678901234567890123      ← A character scale
20 PRINT INSTR(NAM$,";")
30 END
RUN
13
OK
```

INSTR returns the value 13, which, as you can see from the “character scale” in line 15, is the column position of the substring “,” in the parent string NAM\$. The INSTR function in line 20 has two operands: the first defines the parent string in which the search for a character or characters is to be made; the second identifies the substring — the string of characters to be searched for within the parent string.

We could now easily extend this example to return the last name (written with a comma after it), regardless of its length:

```
10 INPUT "last name, first name"; NAM$
20 COL = INSTR(NAM$, ";")
30 PRINT LEFT$(NAM$,COL-1)
40 END
RUN
last name, first name? "Jung, Karl"      ← Enclose name in quotes
Jung
OK
```

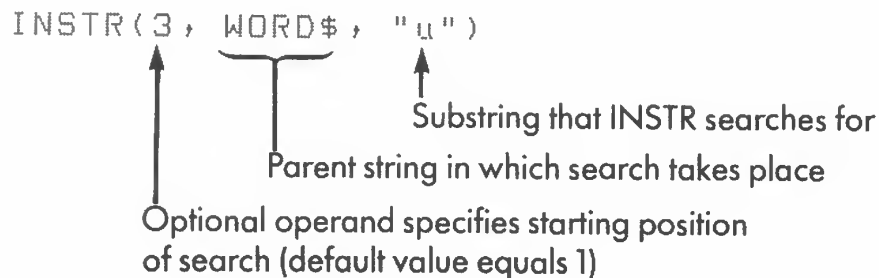
INSTR tells LEFT\$ how many characters to return, starting at the left.

INSTR can search for more than just a one-character string, as the following example illustrates:

```
10 INPUT "Note (do, re, mi, etc)"; NTE$
20 NN = INSTR("doremifasolatiso", NTE$)/2 + .5
30 PRINT NN
40 RUN
RUN
Note (do, re, mi, etc,)? so
5
OK
```

This example shows how to translate a word like *so* into a number, which can then be used for various purposes. In this case, INSTR tells us that "so" is the fifth note in the C-major scale; that information could then be used to play the actual note, as we'll show you in Chapter 16.

In all the examples above, INSTR always started its search at the beginning of the parent string. If we searched for the letter *u* in the Hawaiian word *humuhumunukunukuapuaa* (really, it's the name of a fish!), we'd get the number 2 — the position of the first *u*. But what if we needed to find the position of the second, third, fourth, or even ninth *u*? In cases like this, we must be able to start searching for a particular character at a specified position. INSTR has an optional operand in the first position that specifies the starting position of the search. The following diagram summarizes the three operands of INSTR:



If the value of WORD\$ happens to be the previously named Hawaiian fish, the above function would return the number 4 — the first position of the letter *u* after the specified starting position, column 3.

The Functions LEFT\$, RIGHT\$, MID\$, and INSTR

The following examples illustrate the use of the functions LEFT\$, RIGHT\$, MID\$, and INSTR:

LEFT\$("catnip",3)	← Returns the string 'cat', the left-most three characters
RIGHT\$("catnip",3)	← Returns the string 'nip', the right-most three characters
MID\$("cathair",4,2)	← Returns the string 'ha', which starts in column 4 and is two characters long
INSTR(3,"cathair","a")	← Returns the number 5, the column position of the first a after position number 3

Summary

String functions are an especially interesting group of BASIC instructions. In particular, the functions LEFT\$, RIGHT\$, MID\$, and INSTR provide us with some very sophisticated tools for manipulating strings in any way we might imagine. LEFT\$ and RIGHT\$ extract the left and right portions of a parent string; MID\$ extracts any character or group of characters from a parent string; and INSTR tells us the position of a given character or group of characters within a given parent string.

Other useful instructions, INPUT\$ and INKEY\$, make it possible to input information to a program without pressing **ENTER**. INPUT\$ waits for user response, while INKEY\$ does not.

The remaining functions discussed in this chapter are both useful and easy to use. SPACE\$ generates a string consisting of a specified number of spaces. VAL changes leading digits within a string to a number, and STR\$ converts a number to a string value. STRING\$ generates a string consisting of repetitions of a given character a specified number of times. LEN returns the length of a string.

Exercises

1. Write a program that prints out the number of seconds elapsed since midnight. Use the TIME\$ variable to supply the program with the time in hours, minutes, and seconds. The value of TIME\$ is given as a string constant of the form "HH:MM:SS", where HH stands for hours digits, MM for minutes digits, and SS for seconds digits.

2. Write a program that asks the user to input a note number from 1 to 7 and then prints out the name of the note — do, re, mi, and so on.

Solutions

1. This program prints out the time in seconds:

```
100 REM--NAME:SECNDS,BA-----
110 REM--Prints number of seconds
120 REM--since midnight
130 CLS
140 '--start of GOTO loop-----
150   T$ = TIME$                ← Assigns time as 'HH:MM:SS'
160   HR = VAL(LEFT$(T$,2))      ← Extracts numeric hours value
170   MIN = VAL(MID$(T$,4,2))    ← Extracts numeric minutes value
180   SEC = VAL(RIGHT$(T$,2))    ← Extracts numeric seconds value
190   TSEC = SEC + MIN*60 + HR*3600 ← Total seconds since midnight
200   PRINT @80, "Time =" TSEC " seconds"
210 GOTO 140                    ← Updates calculations
```

This relatively short program manages to use four string functions!

2. The following program prints the note name (do, re, mi, and so on) when the user enters the note number:

```
100 REM--NAME:N>NTE2,BA-----
110 REM
120 CLS
130 INPUT "Note number (1-8)"; NN
140 P = NN*2 - 1                ← Defines starting position
150 NS$ = "doremifasolatio"     ← Parent string
160 NTE$ = MID$(NS$,P,2)         ← Finds desired note
170 PRINT NTE$
180 END
```

This program is similar to the first program in the MID\$ section of this chapter, but in the above program two characters are extracted by MID\$ instead of one.

Sound and Music

Concepts

Pitch, period, frequency
Sound effects
Musical notes
Programming a musical score

Instructions

SOUND

Among the many talents of the Model 100 is its ability to produce sound effects and to play practically any sequence of musical notes. You're already familiar with the BEEP instruction introduced in Chapter 2. BEEP produces a single note of fixed duration and pitch. The SOUND statement, however, is a different matter altogether. By using SOUND, the Model 100 can play a tone of *any* pitch, within a relatively large pitch range, for any specified duration. In the first part of this chapter, we'll introduce the basic SOUND statement, as well as some examples of sound effects that can be created on the Model 100. In the second part of this chapter, we'll see how SOUND can be used to play musical notes and entire musical compositions.

The Basic SOUND Statement

Before we turn the Model 100 into a musical instrument, we should investigate the basic tool for making sounds: the SOUND instruction. Try the following direct command:

```
SOUND 5586, 100
```

The instant you press **ENTER**, you'll hear a tone (this one happens to be concert A) for about two seconds. Experiments in which you change each of the numbers following the word *SOUND* reveal that the first number, which we'll call the *pitch value*, determines the pitch. *Pitch* refers to the high-low quality of a tone: the squeak of a mouse has a high pitch; the blast of a foghorn has a low pitch. The second number, called the *duration value*, determines the length of time the note is played. Both of these numbers can be variables or numeric expressions.

Let's take a closer look at exactly how pitch and duration values affect the output of the *SOUND* statement. Duration is easy to understand: the duration value is directly proportional to the duration of the sound produced. For example, *SOUND 5586, 200* produces a sound twice as long as *SOUND 5586, 100*. The duration value must be in the range 0-255, and a value less than 1 will not result in any audible tone. The maximum value of 255 corresponds to a duration of about five seconds.

To explore the meaning of the pitch value, write the following program and *RUN* it several times for a range of input values:

```
10 INPUT "Pitch value"; P
20 SOUND P,50
RUN
Pitch value? 12000          ← Model 100 plays low-pitched tone
OK
RUN
Pitch value? 800            ← Model 100 plays high-pitched tone
OK
```

You'll notice right away that the *larger* of the two pitch values (12,000 in our first *RUN*) produces the *lower* tone. Be sure to try some other pitch values to get a sense of the range of pitches that your Model 100 can produce. The lowest note you can play has a pitch value close to 16,300; a larger number, such as 16,400, results in an illegal "Function Call Error" (FC Error). The highest audible note results from a pitch value in the neighborhood of 150-200. Though you can use smaller pitch values (for higher tones) without getting an error message, you simply won't hear the note (though you can hear a click) because of the limitations of both your ear and the built-in speaker.

Examples of Simple Sound Effects

The *SOUND* statement can be used to produce various sound effects that might be used effectively as part of other programs, particularly game

programs. The following program makes some sounds that may remind you of computers:

```
10 REM--NAME:R2D2.BA-----
20 REM--Prgm Produces random tones
30 FOR J = 1 TO 50
40   SOUND RND(1)*6200, 1
50 NEXT
```

This program produces a series of fifty very short tones of random pitch values ranging from 0 to 6,200. There may be a few tones with pitch values less than 100 that you'll be unable to hear; these nonaudible tones act as pauses.

The second example steps through a large range of pitch values. The resulting sound may remind you of a falling object or the typical science fiction laser gun:

```
10 REM--NAME:FALLNG.BA-----
20 FOR J = 400 TO 4000 STEP 200
30   SOUND J,1
40 NEXT
```

Changing any of the values in the FOR...NEXT statement will change the character of the sound; for example, a larger step value will speed up the descending "whistle".

The SOUND Statement

We can use the Model 100 to create a tone of a given pitch and duration with a SOUND statement, as illustrated in the following example:

SOUND 2400, 80



Pitch value: specifies the pitch of the tone



Duration value: specifies duration of note

A duration value of 50 corresponds to approximately 1 second; the range of the duration value is 1-255. The pitch value determines the pitch of the note: the *larger* the pitch value, the *lower* the pitch; the range of the pitch value is 16,300 (low note) to approximately 200 (high note).

Using Frequency in SOUND

You already know enough about the SOUND statement to use it to produce various sound effects; you know how to specify the duration of the tone as well as its pitch value. Notice, however, that we haven't been very precise about the meaning of the pitch value. In fact, pitch value is a meaningful quantity only to your Model 100; it is not a quantity that a musician or anyone else working with sound would immediately understand. The most common quantity that specifies pitch is called *frequency*. In this section, we explain the meaning of frequency and how it relates to the pitch value in the SOUND statement.

To understand the meaning of *frequency*, we need to know what *sound* is. All sound begins with a vibration. When we knock on a door, the door vibrates, and this vibration is carried through the air to our eardrums. Whenever we hear a sound, our eardrums are vibrating. The particular pitch we perceive is related to the *rate of vibration*, or the *frequency*. The frequency of a vibration (or sound) equals the number of vibrations in one second. High tones have a high frequency (a large number of vibrations per second), and low tones have a low frequency. Every musical note has a precisely specified frequency. For example, concert A, the note generally used to tune orchestral instruments, has a frequency of 440 vibrations per second. The phrase "vibrations per second" is usually replaced by a standard unit called *Hertz*, abbreviated Hz.

How does frequency relate to the pitch value used in the SOUND statement? Perhaps you've already guessed that pitch value is inversely related to the frequency of the tone; in other words, reducing the pitch value by a factor of two has the effect of doubling the frequency. The following formula translates frequency to the pitch value used in the SOUND statement:

$$\text{Pitch value} = 2457840/\text{frequency}$$

Using this formula, we can write the following program, which asks the user to enter a frequency; the program then prints out the pitch value and plays the corresponding tone:

```
10 REM--NAME:F>SND
20 INPUT "frequency (Hz)";F          ← Frequency F
30 P = 2457840/F                     ← Pitch value P
40 PRINT "pitch value =" P
50 SOUND P, 50
```

To get a sense of how frequency relates to pitch value, as well as the actual sound produced, RUN this program for a wide range of frequencies. In the following two RUNs, we've entered frequencies corresponding to tones close to the lowest and highest that the Model 100 can effectively produce:

```
RUN
frequency (Hz)? 151
pitch value = 16277.086092715      ← Low tone
OK
RUN
frequency (Hz)? 12000
pitch value = 204.82                ← Very high tone
OK
```

Any frequency equal to or less than 150 results in the illegal function call error, and frequencies higher than 12,000 become progressively harder to hear. The number 12,000, which we entered as the frequency, causes a very high and somewhat uncomfortable sound. Musical tones rarely have a frequency above 5,000 Hz.

Using Frequency to Specify the Pitch in SOUND

The following SOUND statement plays a tone having the frequency F:

```
SOUND 2457840/F, 50
```

The lowest frequency F that the Model 100 can play is approximately 151 Hz; the highest audible frequency is approximately 12,000 Hz.

Playing Music

You've seen how to use the SOUND statement to play any tone of a given pitch and duration, and you know that a sequence of SOUND statements can create various sound effects, as illustrated by our previous examples.

However, we can also use the SOUND statement to turn the Model 100 into a real musical instrument. The difference between making sound effects and actually playing music is that music involves very specific frequencies or pitch values. In this section, we will show you how to write programs that play musical notes and complete musical melodies.

There are several approaches to the problem of writing a program that plays music. We'll start with the simplest method, using pitch values, and then develop a more complex program based on the *names* of notes — a program that is more “musician friendly” than the more “computer friendly” program we'll discuss first.

Music with Pitch Values

We'll assume here that you are somewhat familiar with musical notation. As a review and reference, consult Figure 16-1, which shows musical notes on the treble clef and identifies each note by name. (Both the USA standard and Helmholtz notations are used.)

In addition to having a name and position on the musical staff, every musical note is defined by a definite frequency and pitch value. For example, middle C has a frequency of 261.63 Hz and a pitch value of 9,394. Table 16-1 gives the pitch values and frequencies of all the musical notes within the range of the Model 100. (In the exercise section, we'll show you a program that prints out both the pitch values and frequencies of all these notes.) To play music, we need only play a sequence of SOUND statements with the correct pitch values and durations. Though musical notes are usually specified by frequency (rather than by pitch value, which is specific to the Model 100), we'll use the pitch value because it is used directly by SOUND.

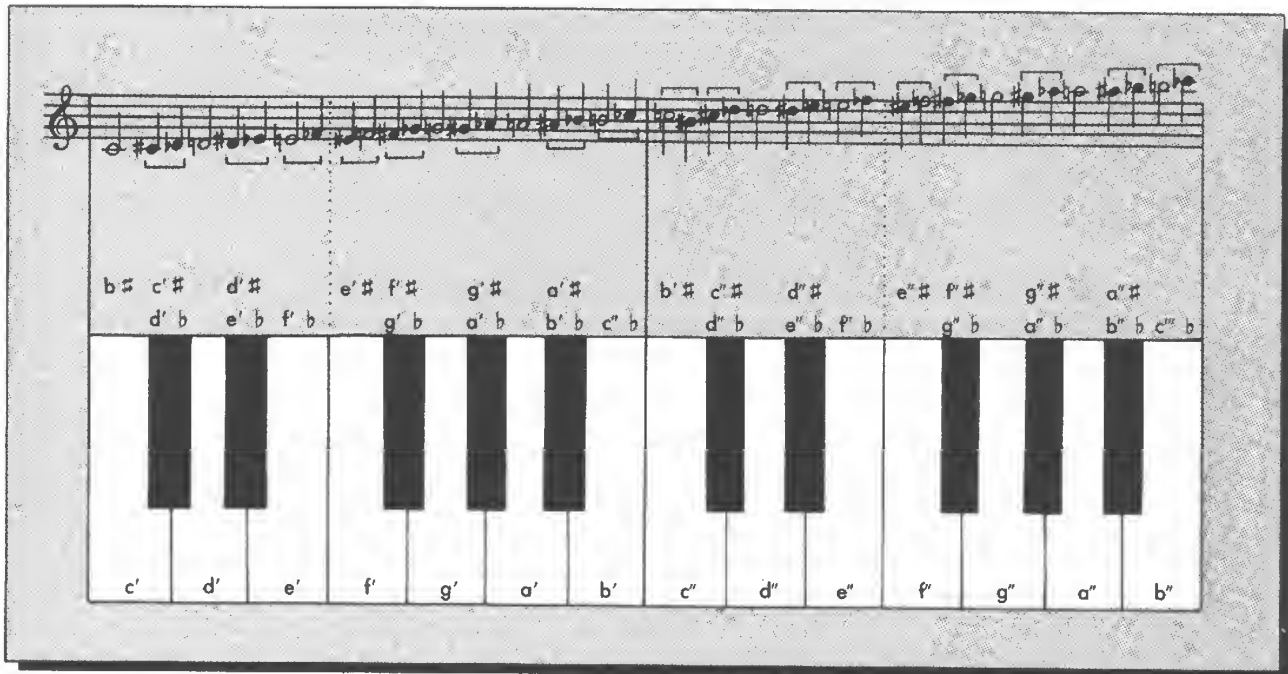


Figure 16-1. Musical notes and the piano keyboard; middle range

Table 16-1. Pitch values and frequencies of the well-tempered scale

Note (USA Standard)	Note (Helmholtz)	Pitch Value	Frequency in Hz
OCTAVE 3			
C3	c	*	*
C3 #	c #,db	*	*
D3	d	*	*
D3 #,E3 b	d #,eb	15799	155.56
E3	e	14912	164.81
F3	f	14075	174.61
F3 #,G3 b	f #,gb	13285	185.00
G3	g	12540	196.00
G3 #,A3 b	g #,ab	11836	207.65
A3	a	11172	220.00
A3 #,B3 b	a #,bb	10544	233.08
B3	b	9953	246.94
OCTAVE 4			
C4	c'	9394	261.63 (middle C)
C4 #,D4 b		8867	277.18
D4	d'	8369	293.66
D4 #,E4 b		7899	311.13
E4	d'	7456	329.63
F4	f'	7037	349.23
F4 #,G4 b		6642	369.99
G4	g'	6270	392.00
G4 #,A4 b		5918	415.30
A4	a'	5586	440.00 (concert A)
A4 #,B4 b		5272	466.16
B4	b'	4976	493.88
OCTAVE 5			
C5	c''	4697	523.25
C5 #,D5 b		4433	554.37
D5	d''	4184	587.33
D5 #,E5 b		3949	622.25
E5	d''	3728	659.26
F5	f''	3518	698.46
F5 #,G5 b		3321	739.99
G5	g''	3135	783.99
G5 #,A5 b		2959	830.61
A5	a''	2793	880.00
A5 #,B5 b		2636	932.33
B5	b''	2488	987.77

OCTAVE 6			
C6	c'''	2348	1046.50
C6 #,D6 b		2216	1108.73
D6	d'''	2092	1174.66
D6 #,E6 b		1974	1244.51
E6	d'''	1864	1318.51
F6	f'''	1759	1396.51
F6 #,G6 b		1660	1479.98
G6	g'''	1567	1567.98
G6 #,A6 b		1479	1661.22
A6	a'''	1396	1760.00
A6 #,B6 b		1318	1864.66
B6	b'''	1244	1975.53
OCTAVE 7			
C7	c''''	1174	2093.00
C7 #,D7 b		1108	2217.46
D7	d''''	1046	2349.32
D7 #,E7 b		987	2489.02
E7	d''''	932	2637.02
F7	d''''	879	2793.83
F7 #,G7 b		830	2959.96
G7	g''''	783	3135.96
G7 #,A7 b		739	3322.44
A7	a''''	698	3520.00
A7 #,B7 b		659	3729.31
B7	b''''	622	3951.07
OCTAVE 8			
C8	c ^v	587	4186.01
C8 #,D8 b		554	4434.92
D8	d ^v	523	4698.64
D8 #,E8 b		493	4978.03
E8	d ^v	466	5274.04
F8	f ^v	439	5587.65
F8 #,G8 b		415	5919.91
G8	g ^v	391	6271.93
G8 #,A8 b		369	6644.88
A8	a ^v	349	7040.00
A8 #,B8 b		329	7458.62
B8	b ^v	311	7902.13

The following program plays four bars of the well-known theme in Beethoven's Ninth Symphony (shown in the usual musical notation in Figure 16-2):

```

100 REM--NAME:MUSIC1,BA-----
110 '--prgm plays musical notes as
120 '--specified in data statements
130 '--beginning with line 1000
140 '
145 DEFINT A-Z
150 READ MX, TP                                ← Reads MX = number of notes & TP = tempo
160 '
170 FOR NTE = 1 TO MX                          ← Plays MX number of notes
180   READ P, D                                ← Reads P = pitch value & D = duration
190   SOUND P, D*TP                            ← Plays note
200 NEXT
210 END
220 '
1000 '==USER-ENTERED DATA=====
1015 '
1020 DATA 15, 2                                ← 15 notes, tempo (TP) = 2
1090 '
1100 'data: each note defined by 2-----
1110 '  numbers: pitch value, duration--
1120 '
1200 DATA 4976,8,4976,8,4697,8,4184,8
1210 DATA 4184,8,4697,8,4976,8,5587,8
1220 DATA 6270,8,6270,8,5586,8,4976,8
1230 DATA 4976,12,5586,4,5586,16

```

You'll probably be impressed when you RUN this program; the Model 100 does a pretty respectable job on Beethoven! Of course, if you prefer a Beatles song or any other music, it's easy to change the DATA statements beginning with line 1000. The first two data entries in line 1020 specify the number of notes to be played (variable MX, with value 15 in our example)

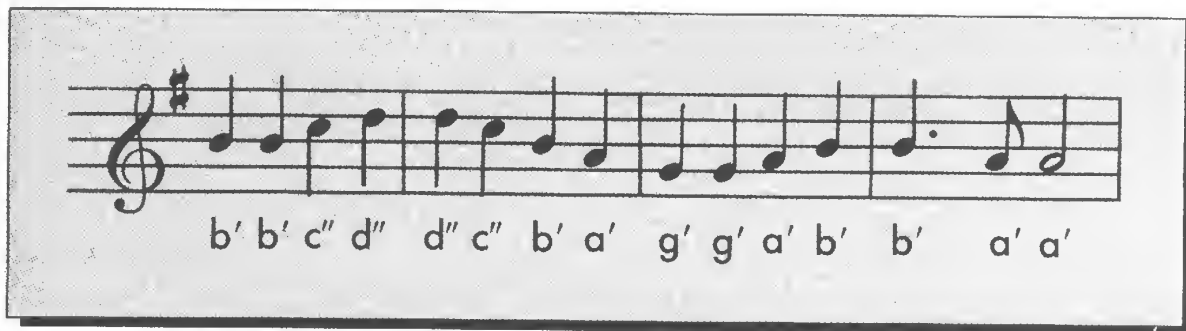


Figure 16-2. Musical score for a theme from Beethoven's Ninth Symphony

and the tempo (variable TP, with value 2 in our example). Increasing the tempo number increases the duration of each note, as you can see from line 190.

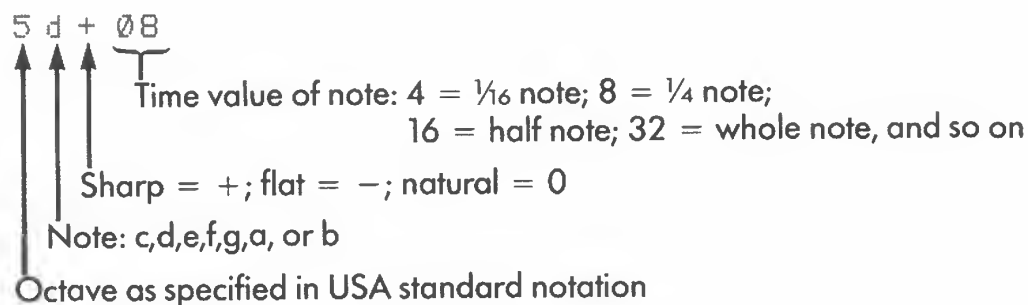
The DATA statements beginning with line 1200 specify the pitch value and duration of each note; that is, each note requires *two* values. For example, the second note, which according to Figure 16-2 is a b', is specified by the pitch value of 4,976 and the duration of 8. We chose the number 8 as a duration value for a quarter note; hence a dotted quarter note (as is the second to the last note) is represented by 12 (one and a half times as long), and an eighth note, by 4 (half as long as a quarter note). If we want to change the tempo of the whole piece, we can simply change the value of TP rather than change the duration value of every note. Note also that in order to help organize our DATA statements, we used one DATA statement for each measure of the musical score.

By now you have all the necessary tools to turn your Model 100 into a musical instrument. The only problem with our previous music-playing program is that the notes of a musical composition must be specified in a nonstandard code that uses pitch values. In the following section we describe how to bypass the need to use Table 16-1 to find pitch values for given musical notes. We'll develop a program that allows the user to specify notes by the standard letter codes; the program then *calculates* the pitch values of the notes to be played.

A More Sophisticated Program to Play Music

We want to develop here a music-playing program that allows the user to enter a melody in a notation similar to standard musical notation.

Before attempting the final version of a program that plays a whole melody, let's begin with a program that plays a single note entered by the user. The note should have the form illustrated in the following example:



The indicated time values are somewhat arbitrary; what is important is that different notes receive time values proportional to their intended length. If the actual time values listed above are used in the SOUND statement, the resulting tempo of the notes will be approximately equivalent to the musical notation *adagio* (slow).

A program that is to play a note specified in the manner illustrated above must perform the following sequence of operations:

1. Request user input of the note to be played (N\$).
2. Extract the octave value (OCTAVE), the note value (NO\$), and the time value (T) from the user-entered string N\$.
3. Convert the note value (NO\$) and octave value OCTAVE to note number (NN).
4. Use the note number (NN) to calculate the frequency F of the note. The frequency ratio of adjacent notes of the well-tempered scale is given by the twelfth root of 2, or $2^{(1/12)}$.
5. Play the note using the SOUND statement, which converts frequency to pitch value.

As illustrated above, when tackling a program as complicated as this, it helps to break it down into a sequence of simple, manageable tasks. Here's our program:

```

100 REM--NAME:MUSIC2.BA-----
110 'prgm requests input of note, then
120 'plays note
130 '
140 '--some needed constants-----
150 '
160 C4 = 261.63                                ← Freq. of C4
170 S1$ = "c0c+d0d+e0f0f+g0g+a0a+b0"         ← 12 note names, naturals & sharps
180 S2$ = "c0d-d0e-e0f0g-g0a-a0b-b0"         ← 12 note names, naturals & flats
190 '
200 '--main Program-----
210 '
220 INPUT "Note, as in 4F+08";N$                ← Step 1: input of note
225 '
230 OCTAVE = VAL(LEFT$(N$,1))                  ← Step 2: extract octave
240 NO$ = MID$(N$,2,2)                         extract note name
250 T = VAL(RIGHT$(N$,2))                     extract time value
260 '

```


370 NN = (INSTR(1,S1\$,NO\$))/2% - .5	← Step 3: calculate note number NN;
380 IF NN <> 0 THEN 410	if found, cont. with 410;
390 NN = (INSTR(1,S2\$,NO\$))/2% - .5	try S2\$ in case of flat
400 '	
410 F = C4*2^(OCTAVE - 4 + NN/12)	← Step 4: calculate frequency F
420 '	
430 SOUND 2457840/F, T	← Step 5: play note
440 END	

String functions really come in handy in this program! Though most of the program is readily comprehensible, a few statements toward the end of the program need some clarification. The function of lines 370, 380, and 390 is to find NN, the position of the string NO\$ in the parent string S1\$ or S2\$; S1\$ contains only sharps and S2\$ contains only flats. The value of NN tells us the number of half steps between the entered note and the note C in the same octave. Line 410 then uses this value of NN and the octave number to calculate the frequency of the note. The expression in parentheses in line 410 finds the number of half steps between the note C4 and the user-entered note. The frequency equals the frequency of C4 times two, raised to a power equal to the number of half steps.

Now we're ready to tackle the complete music-playing program. The main feature we need to add to our previous program is the ability to play a sequence of notes rather than just a single note. Rather than asking the user to enter each note in response to an INPUT statement, a process which would have to be repeated every time the program is run, we'll request the user to enter the melody as strings in a series of DATA statements. Also, because our program is fairly slow (because a fair number of BASIC statements are involved in processing each note), we'll first load all the calculated values for pitch and duration into two arrays, called *PV* ("Pitch Value") and *DU* ("DUration"), and then play the notes all at once. In this manner, we avoid the excessive pauses between adjacent notes that would inevitably result if the program had to determine the pitch value and duration of each note just before playing the note.

The following is our final music-playing program. You can use our previous program, MUSIC2.BA, as a starting point for this program because lines 160-180 and 330-410 are the same in both. The data strings at the end of the program define the theme from Beethoven's Ninth Symphony that we used before.

```

100 REM--NAME:MUSIC3.BA-----
110 'Prgrm plays melody specified by
114 'data strings beginning with
116 'lines 2000
118 '
120 CLS
122 PRINT "--Please wait--"
124 '
140 '--some needed constants-----
150 '
160 C4 = 261.63                                ← Freq. of C4
170 S1$ = "c0c+d0d+e0f0f+g0g+a0a+b0"         ← 12 note names, naturals & sharps
180 S2$ = "c0d-d0e-e0f0g-g0a-a0b-b0"         ← 12 note names, naturals & flats
190 '
200 '--read NU = number of notes
202 '      ND = number of strings
204 '      TEMPO = tempo of music----
206 '
210 READ NU, ND, TEMPO
215 '
220 DIM PV(NU), DU(NU)
225 '
230 '--stores pitch value and duration
235 ' for each note in PV() & DU()---
240 '
250 FOR S% = 1 TO ND
260   READ NTE$                                ← Reads ND number of strings
270   L = LEN(NTE$) + 1                        ← Effective length of each string
280   FOR J% = 1 TO L-5 STEP 6                 ← Processes each note
290     N$ = MID$(NTE$,J%,5)                  ← Extracts note from NTE$
330     OCTAVE = VAL(LEFT$(N$,1))             ← Extract OCTAVE from N$
340     NO$ = MID$(N$,2,2)                   ← Extract note name
350     T = VAL(RIGHT$(N$,2))                ← Extract time value
360   '
370     NN=(INSTR(1,S1$,NO$))/2%-.5           ← Calculate note number, NN
380     IF NN <> 0 THEN 410                   ← If found, cont. with 410
390     NN=(INSTR(1,S2$,NO$))/2%-.5           ← Try S2$ in case of flat
400   '
410     F = C4*2^(OCTAVE - 4 + NN/12)         ← Calculate frequency F
420   '
440     C = C + 1                             ← Note counter
450     PV(C) = 2457840/F                     ← Loads PV() with pitch value
460     DU(C) = T * TEMPO                    ← Loads DU() with duration val
470   NEXT J%
480 NEXT S%

```

```

490 '
500 '--Plays notes-----
510 '
515 PRINT "OK, here we go,.,."
520 FOR J% =1 TO NU
530   SOUND PV(J%), DU(J%)
540 NEXT
550 END
560 '
1000 '==USER-ENTERED DATA=====
1002 '
1004 '--Title: Beethoven's 9th Symphony
1010 '
1020 '--data values NU = total # of notes
1030 '   ND = number of strings with notes
1040 '   TEMPO = tempo: 4 is Legato, approx
1050 '
1060 DATA 15,4,2
1070 '
1100 '--data values: notes as in 6d016-----
1110 '
1120 DATA "4b008,4b008,5c008,5d008"
1130 DATA "5d008,5c008,4b008,4a008"
1140 DATA "4g008,4g008,4a008,4b008"
1150 DATA "4b012,4a004,4a016"

```

This is a rather long program, and it takes a few seconds to get to the point of actually playing the melody — but then it does a beautiful job. The primary advantage of this program is that we can enter the notes easily and rapidly once we know the octave number of a note, its name, and its duration. This program also uses pretty much everything you've learned about BASIC!

Summary

You've finally done it — your Model 100 can play music! Though we introduced only one statement in this chapter, the SOUND statement, you've learned to use it in many different ways. Used in the simplest way, it can play a tone of a given pitch and duration if the *pitch value* and *duration value* are specified. For some applications, it is more natural to specify the *frequency* of the tone; a simple conversion formula can be used to translate a frequency value to a pitch value.

1. Write a program that makes one ticking sound approximately once per second.
2. Write a program that requests the user to enter a note using the notation do, re, mi, and so on, and then plays that note.
3. Write a program that plays each note in four octaves of the well-tempered scale, beginning with A3; it should also print out the pitch value and frequency of each note after it is played.
4. Rewrite the DATA statements in program MUSIC3.BA to play the following melody:



1. This program makes a tick approximately once per second:

Sound and Music 301

2. This program plays a note specified by user input as do, re, mi, and so on:

```
100 REM--NAME:DOREMI.BA-----
110 REM--Plays notes do, re, mi, etc.
120 REM--in 4th octave
130 REM
140 C4 = 261.63 'freq. of C4 or c'
150 INPUT "Note as do,re,mi,etc."; NTE$
160 NN = (INSTR(1,"doremifasolati",NTE$)-1)/2
170 F = C4*2^(NN/12)
180 SOUND 2457840/F, 50
190 END
```

3. The following program plays four octaves of the chromatic scale, beginning with A3 at 261.63 Hz, and prints out pitch values and frequencies:

```
100 REM--NAME:CHRMSC.BA-----
110 REM--Prgm Plays chromatic scale
120 REM
130 RATIO = 2^(1/12) ← Frequency ratio between adjacent notes
140 F = 220 ← Frequency of A3, exact
150 P = 2457840/F
160 '
170 PRINT "P value"; " Freq,"
180 PRINT
190 '
200 FOR NTE = 1 TO 48
210 PRINT CINT(P);
220 PRINT USING "#####.##";F
230 SOUND P,10
240 F = F*RATIO ← Finds next value of frequency F
250 P = 2457840/F ← Finds next pitch value P
260 NEXT
```

This program prints out the pitch values and frequencies shown in Table 16-1, though over a slightly smaller range.

4. These DATA statements should be substituted for the DATA statements in MUSIC3.BA to play “Oh du lieber Augustien”.

```
1000 '--user-entered data-----
1060 DATA 24,8,1
1120 DATA "5c020,5d008,5c008,4b-08"
1130 DATA "4a016,4f016,4f016"
1140 DATA "4g016,4c016,4c016"
1150 DATA "4a016,4f016,4f016"
1160 DATA "5c020,5d008,5c008,4b-08"
1170 DATA "4a016,4f016,4f016"
1180 DATA "4g016,4c016,4c016"
1190 DATA "4f064"
```

Data Files

Concepts

- Data files for information storage
- Writing and appending data files
- Reading data files
- Records with multiple fields
- Using TEXT to write a data file
- Sending data to screen, printer, and other computers

Instructions

- OPEN...FOR OUTPUT, OPEN...FOR APPEND, OPEN...FOR INPUT, MAXFILES, PRINT #, CLOSE, INPUT #, EOF, TEXT

You've seen how BASIC on your Model 100 can manipulate numbers and strings, make decisions, draw images, and play music. BASIC can also manage the flow of information from one location to another. Whenever we use CSAVE, for example, we're sending information (a program) from random access memory (RAM) to a cassette file. Whereas the user controls this particular process, a BASIC *program* can also manage the flow of information in the form of *data files*. A data file, for example, might contain the pitch and duration values of musical notes, the items of an expense account, or the contents of a letter. BASIC programs that manage such data files can transmit the file to any of several different locations, such as RAM memory, a cassette or disk file, or a *modem*. (A modem is an electronic device connected to a computer and a telephone line for the purpose of transmitting a file over the telephone.) A BASIC program can also "read" or receive a data file; the file can then be displayed or processed in some manner. In this chapter we explore data files, including how to transmit and receive them.

There are two general purposes for using data files. One is as a *storage medium* for information. The Model 100 has three data-storage locations: the most convenient is RAM; for more permanent storage, data files can also be stored on cassette tapes or floppy disks, provided that you have the necessary hardware.

The second purpose of using data files is to *transmit* information (the file) from one location to another. For example, a BASIC program can send a RAM file (file stored in random access memory) to the printer. Or it can receive information transmitted over the telephone and store it in RAM. In the last part of this chapter, we will introduce such transmission files, though we don't have space to cover all aspects of this topic.

Many examples used early in this chapter are about RAM files. RAM files are the easiest to use and also the most useful for many Model 100 users. Because all the different files can be treated in much the same way, everything you learn about RAM files can be applied to other types of files.

Storing Information in a Data File

In Chapter 16 we used DATA statements to store musical notes. The DATA statement is ideal for storing permanent information within a program. Musical notes, however, are not necessarily permanent; we may wish to change notes or even the whole composition. Instead of storing musical information using DATA statements, we can place this information in a data file separate from the program that actually plays the notes. We can easily modify or edit data files, which makes them ideal for storing flexible information.

Data files are fairly complex, so let's begin with a simple example: using a BASIC program to create as well as to read a RAM data file. Later, we'll explore some of the more sophisticated variations of data files.

Writing and Reading a Simple Data File

Data files can be created and read by a BASIC program. If we think of a data file as a filing cabinet, then the program that creates and reads these files is the "secretary" that manages them. Before presenting and explaining all the new statements needed in such file-management programs, we

suggest that you enter the following complete program in order to get an overview of the whole process of creating and reading a file:

```

100 '--NAME:SQUARS.BA-----
105 CLS
110 '--creates data file-----
120 '
130 OPEN "RAM:SQR.DO" FOR OUTPUT AS 1
140 FOR J = 1 TO 5
145     SQUARE = J * J
150     PRINT #1, SQUARE
160 NEXT
170 CLOSE 1
180 END
190 '
200 '--reads data file-----
210 '
220 OPEN "RAM:SQR.DO" FOR INPUT AS 1
230 FOR J = 1 TO 5
240     INPUT #1, SQUARE
250     PRINT SQUARE;
260 NEXT
270 CLOSE 1
280 END

```

The first part of this program (up to line 180) creates a file called "RAM:SQR.DO", which contains the squares of the numbers 1-5. The second part (beginning with line 200) reads this file. Though we could have written two separate programs — one for writing the file and one for reading it — we combined both of these functions into a single program so that we could read the file right after we created it, a good way to find out what actually ended up in our file!

Now RUN this program. Shortly after pressing RUN, the Ok prompt will appear on your screen. Presumably, you now have a data file called "SQR.DO" in RAM. To see if this is true, RUN the second half of the program by entering CONT. CONT stands for CONTinue; it tells the Model 100 to continue running the program where it left off. (You could enter RUN 100 instead.) The following should now appear on your screen:

RUN		← Model 100 executes lines 100-180
OK		
CONT		← CONTinues execution of lines 190-280
1	4 9 16 25	← Contents of file 'RAM:SQR.DO'
OK		

Sure enough, the second half of the program correctly read the contents of the file created by the first part — the square of the numbers 1-5. You can also return to the main menu to see that the file SQR.DO is indeed listed as one of your files. Let's take a look at some of the details.

The OPEN Statement

Before we can put something into a filing cabinet, we must open a file drawer. The OPEN statements in lines 130 and 220 fulfill a function analogous to opening a filing cabinet. Until we've opened a file in this manner, we won't have access to what's inside. Let's see exactly what each part of the OPEN statement means.

The phrase "RAM:SQR.DO" in quotation marks after OPEN is sometimes called the *file specification*. You've already used such file specifications when saving and loading BASIC programs. The prefix RAM specifies that the file being opened is a "RAM data file". That is, data are to be stored or retrieved from RAM. Two other possible data-storage locations are the tape cassette and the floppy disk. *Cassette files* must be prefixed by the device specification CAS; *disk files* must be prefixed by the device specification 0 or 1, depending on whether disk drive 0 or 1 is being used. If no prefix is indicated, BASIC assumes we mean to specify a RAM file. Several other types of data files (and prefixes) exist; we'll introduce these later in this chapter.

Following the prefix RAM and the colon is the filename SQR.DO. Rules for data filenames are identical to those for BASIC programs, except that data files must have the suffix .DO rather than .BA. Files ending in .DO are called "DOcument" or "text" files; we'll explain more about document files when we discuss using the TEXT editor to create data files. Examples of possible data file specifications are shown below:

RAM:NOTES4.DO	
CAS:STOCK.DO	
RECEPE.DO	← A RAM file by default
0:GRADES.DO	← Data file GRADES to disk drive 0

Note that file specifications must always appear within quotes following the SAVE or LOAD commands.

The next part of the OPEN statement, "FOR OUTPUT" in line 130 and "FOR INPUT" in line 220, informs the Model 100 of the purpose for opening the file. The statement

```
130 OPEN "RAM:SQR.DO" FOR OUTPUT AS 1
```

instructs BASIC to OPEN a file in order to move or OUTPUT information *from* the program *to* the file. This part of the OPEN statement *creates* a file.

Similarly, the statement

```
220 OPEN "RAM:SQR.DO" FOR INPUT AS 1
```

instructs BASIC to OPEN a file in order to move or INPUT information *to* the program *from* the file. This part of the OPEN statement opens an existing data file in order to *read* it.

The last part of the OPEN statement, "AS 1" in our example, specifies the file number. Though our program has only one file open at any given time, we can open more than one file. In that case, the file number is a convenient reference to a particular file in file-related statements that follow the OPEN statement. For example, the number 1 in the statement PRINT #1 refers to the file RAM:SQR.DO by referring to the file number rather than the filename. The file number must be included in all file-related statements (the only exception is the CLOSE statement, as we'll see later), even though only one file may be open.

If we want to have more than one file open at any given time, we must inform the Model 100 of our intentions before actually opening the files by reassigning a special variable called MAXFILES. MAXFILES has a value equal to the "MAXimum" number of "FILES" that we can open. The default value is 1, which means that unless this variable is reassigned, only one file can be open at any given time. To reset MAXFILES to, say, 4, we use the usual assignment statement:

```
30 MAXFILES = 4
```

We can subsequently have up to four files open at the same time. The maximum value of MAXFILES is 15.

Though MAXFILES is a *reserved variable* (reserved by BASIC for the purpose of specifying the maximum number of files), we can treat it like other variables. For example, we can print out its value with the statement "50 PRINT MAXFILES".

Here's an important point: a file can only be open for both input and output at the same time if it has been given *two different* file numbers. In our example program, notice that we first "closed" (by the CLOSE statement in line 170, which we'll explain later) the file opened FOR OUTPUT AS 1 before we opened the file FOR INPUT AS 1 in line 220.

The OPEN Statement

This variation of the OPEN statement is used to *create* a file:

File number that is subsequently associated with the filename →

```
30 OPEN "RAM:STOCK.DO" FOR OUTPUT AS 2
```

File specification: destination, filename, and extension; default destination is RAM

OUTPUT from program to file

To *read* the file created by the above statement, use the statement

```
40 OPEN "RAM:STOCK.DO" FOR INPUT AS 1
```

where the word *INPUT* implies “INPUT to the program *from* the file”. Only one file may be open at any given time unless the reserved variable MAXFILES is first reassigned to the number of files that will be opened.

The PRINT # Statement

Once we open a file for output, we can write information to the specified file by means of the PRINT # statement. (Note that this is a different statement than the familiar PRINT.) For example, line 150 in our previous program,

```
150 PRINT #1, SQUARE
```

gives the instruction “print the value of SQUARE into file #1”. The number 1 in “#1” is the same file number specified in the OPEN statement in line 130. The value that’s written to a file (the value of SQUARE, in this example) is sometimes called a *record*.

Writing to a file with PRINT # creates a file image similar to what appears on the screen if we use PRINT. The main difference between the

two is that PRINT sends information to the screen, whereas PRINT # sends information to a file. Figure 17-1 illustrates the relationship between PRINT and PRINT #.

We can also write strings directly to our file in the same way that we'd PRINT a string onto the screen. For example, we can replace line 150 with the statement

```
150 PRINT #1, "raspberries"
```

in order to write the word *raspberries* to the file.

We can also use a single PRINT # statement to write records consisting of more than one variable (we'll return to this topic later).

The CLOSE Statement

When we're finished with a particular file in a filing cabinet, it's customary to close the file drawer. That's what the CLOSE statement does in a BASIC data file. If we don't close our files with CLOSE, it is possible to lose data or to alter it in an unexpected manner. Line 170 in our program

```
170 CLOSE 1
```

says "CLOSE the file opened under the file number 1". Notice that we used the same CLOSE statement in lines 170 and 270 — CLOSE doesn't care whether the file was originally opened for input or for output. We can also use a single CLOSE statement to close several files. For example,

```
300 CLOSE 1,3
```

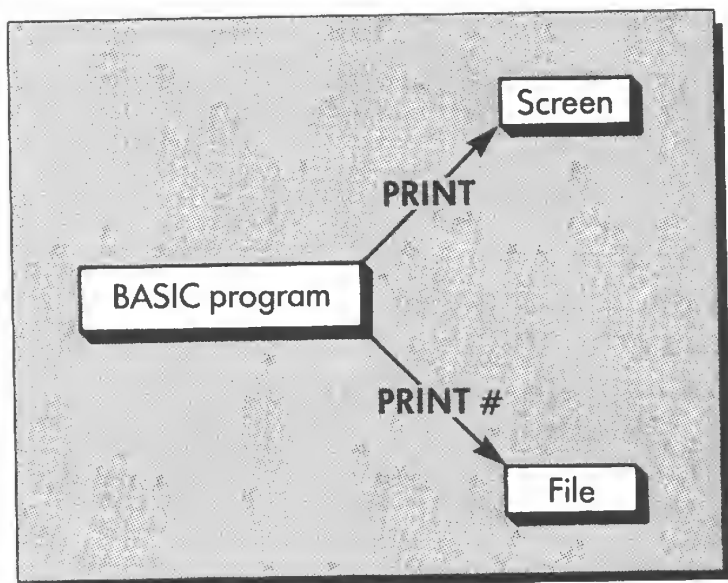


Figure 17-1. Relationship between PRINT and PRINT #

closes files number 1 and 3. If we omit the file number altogether, CLOSE closes all files. Because our program had only one file open at a time, we could have used CLOSE instead of CLOSE 1. Actually, we could have omitted CLOSE in line 270, because END automatically closes all files. Nevertheless, it is good programming practice to explicitly close files in order to clarify program flow and to prevent any chance of messing up or losing a data file.

The INPUT # Statement

You've seen that PRINT writes information to the screen, whereas PRINT # writes information to a file. The same relationship exists between INPUT and INPUT #. INPUT # reads data from the specified file to the program. For example, line 240 of our program

```
240 INPUT #1, SQUARE
```

inputs a number stored in the file opened under file number 1 and assigns this number to the variable SQUARE. Figure 17-2 illustrates the relationship between INPUT and INPUT #.

The PRINT #, INPUT #, and CLOSE Statements

Records are read to the file and retrieved by the following two statements:

```
120 PRINT #2, X
```

← Writes the value of X to the file opened under file number 2

```
220 INPUT #3, ADDRESS$
```

← Reads or inputs the value of ADDRESS\$ from the file to the program

Files are closed by a CLOSE statement. The following are several variations:

```
CLOSE 1,2
```

← Closes files identified by file numbers 1 and 2

```
CLOSE
```

← Closes all files

Keeping Order in Reading and Writing

In our example program SQUARS.BA, values in the file RAM:SQR.DO are read and assigned to SQUARE in the same order that they were originally written to the file. For example, the second time INPUT # is executed (for J = 2), it reads the second record in the file, which is 4. Data must be read in the same order as originally written. Because information is written and read in the same sequence, this type of file is often referred to a *sequential file*. We might imagine a sequential file to be like the storage bin shown in Figure 17-3, into which records are entered from above and retrieved from below. Records are stored in a definite sequence, without any mixing — the first record in is always the first record out. This is very much like a type of investment called “first in, first out”, sometimes abbreviated FIFO.

A Second Look at Data Files

Data (or sequential) files are a large and complex subject. So far we’ve covered the bare essentials of using data files for the particular purpose of storing information. In the following sections we’ll take a closer look at some of the more sophisticated aspects of data files. In particular, you’ll learn how to append data files, how to read a whole data file without the need to know how many records it contains, how to use records containing more than one value, and how to create a data file with the TEXT editor. We’ll also introduce the use of data files to send information to devices other than

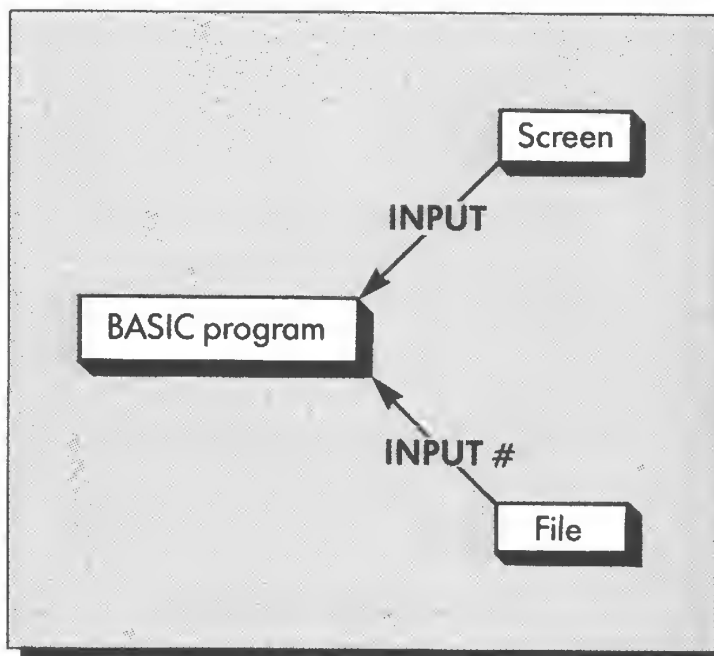


Figure 17-2. Relationship between INPUT and INPUT #

random access memory (RAM) and cassette tapes. In the process, we'll also show you several interesting and useful examples.

Adding Data to a Data File

We often need to add data to an existing file. For example, if we wanted to maintain an up-to-date file of stock prices, we'd periodically need to add the most recent stock prices. Simple! We just open the file FOR OUTPUT and start adding data, right? Wrong. The problem is that as soon as we open a data file FOR OUTPUT, we destroy its current contents. The solution is to use another variation of the OPEN statement, called OPEN...FOR APPEND, which is specifically designed to append an existing file.

As an example, let's write a program that creates a file of daily stock values of a given company:

```
100 REM--NAME:STOCK1.BA-----
110 REM--Prgm allows user to append
120 REM--to file of stock values
130 CLS
140 OPEN "RAM:TANDY.DO" FOR APPEND AS 2
145 '--beginning of GOTO loop-----
150 INPUT "stock value (0 if finished)"; ST
160 IF ST = 0 THEN 200
170 PRINT #2, ST
180 GOTO 145
200 CLOSE
210 END
```

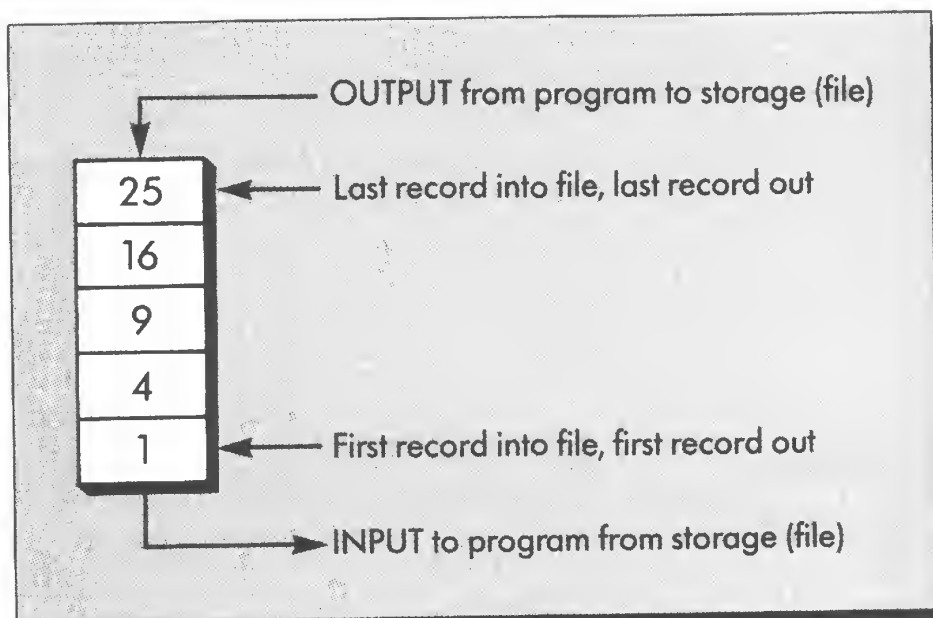


Figure 17-3. Data storage in a sequential file

This program will continue to load new values of ST into the file "RAM:TANDY.DO" as long as we keep entering values other than 0. If we enter 0, the data file will be closed by the CLOSE statement in line 200 and we'll be back in the BASIC Command Mode.

The new statement is the OPEN...FOR APPEND statement in line 140. Its effect is identical to that of OPEN...FOR OUTPUT, except that OPEN...FOR APPEND causes a file to be *appended*, or added to, whereas OPEN...FOR OUTPUT writes *over* an existing file. Every time we run the above program STOCK1.BA, stock values entered will be *added* to the existing file. If no file of the specified filename exists, OPEN...FOR APPEND creates a new file. On the other hand, if we used OPEN...FOR OUTPUT in place of OPEN...FOR APPEND in line 140, the existing file would first be deleted before new values could be entered. Well, that's fine for certain applications, but when we're trying to compile a list of numbers (or strings) over a period of time, we don't want to have to start over from the beginning each time we add new values.

The OPEN...FOR APPEND Statement

Whenever we need to *add* data to an existing data file, we use the following type of statement to open the file:

```
100 OPEN "CAS:STOCK4.DO" FOR APPEND AS 1
```

OPEN...FOR APPEND adds new data to the end of the existing file, in contrast to OPEN...FOR OUTPUT, which writes over the specified file.

The EOF Statement — Knowing When to Stop

Our first example program, SQUARS.BA, used a FOR...NEXT loop to read the file RAM:SQR.DO. This technique works well as long as we know in advance exactly how many records exist in the file. But if we don't know how many records exist, we can run into a problem: though it's perfectly Ok not to read the whole file — no error messages appear, we just get what we asked for — it is not legal to try to read *more* than what actually exists in a file. If we try to read more records than actually exist, we'll get the error message

```
?EF Error in 140
```

assuming that line 140 contains the INPUT # statement. The letters *EF* stand for “End of File”; the error message tells us that we tried to read past the end of the file.

It would be helpful to always know when we get to the end of a file, so we wouldn’t have to be concerned with how many records we need to read. Such a feature would be especially helpful if we were trying to read a file of variable length, such as the file of stock prices RAM:STOCK.DO created in the last section. Well, what we’re looking for is the EOF function.

EOF stands for “End Of File” — it informs the program that is reading a data file when the end of the file has been reached. We can use this function as part of an IF..THEN statement, as shown in the following example, which reads the file RAM:STOCK.DO created by the previous program, STOCK1.BA, and finds the average stock price (if we wish, we can simply add these lines to STOCK1.BA):

```
300 '--Prgm reads complete file
310 '  RAM:STOCK.DO-----
320 '
325 N% = 0          ← Initializes N% = number of records read
330 T = 0          ← Initializes T = sum of stock values read
335 '
340 OPEN "RAM:STOCK.DO" FOR INPUT AS 1
345 '
350 '--beginning of GOTO loop-----
360   INPUT #1, ST
370   PRINT ST
380   T = T + ST      ← Total of stock values read
390   N% = N% + 1     ← Number of stock values read
400   IF EOF(1) THEN 420 ← Goes to 420 if end of file
410 GOTO 350
420 CLOSE
430 PRINT "stock average ="; T/N%
440 END
```

An English translation of the EOF statement in line 400 is: “IF the End Of the File #1 has been reached, THEN go to line 420”. Line 420 closes the file. Perfect! The EOF statement allows our program to read exactly what’s in the file — nothing more, nothing less! It’s a good idea to read *all* files with the help of the EOF function.

The EOF Variable

EOF tests for the “End Of a File”, as illustrated in the following statement:

```
340 IF EOF(2) THEN 400
```

which means “if the end of the file has been reached, then go to line 400”.

Records with Multiple Fields

In our examples, we have used PRINT # and INPUT # to read one number or one string at a time. But we can also write and read more than one item with each PRINT # and INPUT # statement. Writing and reading files in this manner is very useful, for example, when we want to store information that is naturally organized in pairs of values, such as the pitch and duration values of musical notes.

The following single PRINT # statement writes *two* numbers, the pitch value and the duration value of a note, to a file named in a preceding OPEN...FOR APPEND statement:

```
70 PRINT #1, PITCH, DU
```

The pair of values of PITCH and DU is called the *record*; each data item within the record is called a *field*. This particular record has two fields, defined by the variables PITCH and DU. More than two fields are also possible — the only real limit is the 255-character length of a BASIC line.

Not surprisingly, a record consisting of multiple fields can also be read by a single INPUT # statement. For example, the statement

```
140 INPUT #1, PITCH, DU
```

reads two values from the file named in OPEN...FOR INPUT and assigns them to the variables PITCH and DU. This statement can be used to read the file written by the previous PRINT # statement, though we don't need to use the same variable names.

We can also use multiple fields containing strings or a mixture of strings and numbers, but there is one minor complication, which may result in an

EF error message unless we take special steps to avoid it. The problem, for example, is that the line

```
120 PRINT #1, CO$, STCK
```

writes the values of CO\$ and STCK to a file as a *single string value*. We can see this if we then try to read the values of CO\$ and STCK from the file with a corresponding INPUT # statement; we'll get an EF error message because only one value exists — a string consisting of the value of CO\$, some spaces, and the digits in STCK. Such merging of two apparent fields in a record happens whenever the first field contains a string value. Not everything in the computer world makes sense, and this is one that doesn't.

Fortunately, there is a simple solution to this anomaly. We need only insert a string consisting of a comma after every string in the record. For example, the statement

```
120 PRINT #1, CO$, ",", STCK
```

will correctly write two separate values to a file so that there will be no problem in reading and printing out this record with the following statements:

```
250 INPUT #1, CO$, STCK
260 PRINT CO$;STCK
```

Notice that we used a semicolon in the PRINT statement; if we had used a comma, the values of CO\$ and STCK would be printed out on *different* lines. The reason for this behavior is that the comma we introduced in the PRINT # statement causes an extra carriage return. If we want the values of CO\$ and STCK to be printed out on the same line, as we did, we must use a semicolon.

Using TEXT to Write a Data File

As you've seen, the names of all data files must end with the suffix DO. DO stands for "DOcument", and files with this suffix are called *document* or *text* files. In contrast, BASIC program files are usually saved under a filename ending in BA. Though BASIC programs and data files both constitute information, this information is stored differently in the memory of the Model 100. BASIC programs filed under a BA suffix are stored in a special code unique to the Model 100, whereas document (DO) files are stored using ASCII code values.

The significance of the fact that data files are document files is that we can easily create document files with the Model 100's built-in editor, called

TEXT. In other words, instead of writing a data file using a BASIC program the way we've done, we can conveniently write a data file using TEXT. A data file created in this manner can be read by a BASIC program in exactly the same manner as we've already described.

To see how to use TEXT to write a data file, let's rewrite the program MUSIC1.BA introduced in Chapter 16, which plays a theme of Beethoven's Ninth Symphony. First, we'll write the pitch and duration values of each note into the data file using TEXT (rather than using DATA statements to store them); then we'll read this file and play the notes with a BASIC program.

First, we need to know how to use TEXT. That's easy: we return to the main menu of the Model 100, place the cursor over the file called TEXT, and press **ENTER**. The following question immediately appears on the screen:

```
File to edit?
```

Now we reply with a filename no more than six letters long, say, "BTHN9". The suffix DO is optional. The screen clears once more except for a blinking arrow in the upper left corner — the TEXT prompt. Its function is similar to the BASIC prompt in that it tells us where a character will appear if we press a key. Now we're ready to enter the pitch and duration values of our musical theme.

To enter pitch and duration values, we simply type the same values used in the data statements of MUSIC1.BA. These values can be written in many different ways; below is one possibility:

```
4976,8  4976,8  4697,8  4184,8
4184,8  4697,8  4976,8  5587,8
6270,8  6270,8  5586,8  4976,8
4976,12 5586,4  5586,16
```

Each pair of numbers represents one note: the first number is the pitch value, the second, the duration value. We've separated pitch and duration values with commas and inserted one or two spaces between each pair of values representing each note; also, each line contains all the notes for one measure. This particular format, however, is just one possibility. For our BASIC program to be able to read this file, at least one space or a comma must exist between two separate numbers; it's that simple. This formatting requirement leaves us a great deal of latitude in visually organizing data on the screen. For example, we can use one line for each note, or we can cram all data into as tight a space as possible, using only commas or single spaces

to separate each number. This versatility in formatting data is one of the really convenient features of writing data files using TEXT.

What if we make a typing mistake or simply want to change a data value? Making corrections or additions could hardly be simpler, because TEXT uses the same editing commands as the BASIC editor (invoked by entering EDIT in the BASIC Command Mode), described in Chapter 4. For example, we can use the four cursor control keys, as well as the **DEL/BKSP** key, in exactly the way we used them in the BASIC editor.

Leaving the TEXT mode is identical to leaving the BASIC editor; we simply press **F8**; the only difference is that when leaving TEXT, we'll get back to the main menu rather than our BASIC program. As you can see from an inspection of the main menu, our file has been saved as the document file BTHN9.DO. To save a document file to cassette tape or to disk, press **F3**, which stands for "SAVE" (press the **LABEL** key to identify function keys if you need to be reminded), and enter the file specification. For example, to save our previous list of pitch and duration values to a disk file called BTHN9, we'd enter the file specification 0:BTHN9.DO so that the bottom line on the screen would look like this:

```
Save to: 0:BTHN9.DO
```

To save BTHN9 as a cassette file, simply replace the device specification 0 by CAS.

All right, we've got our file called BTHN9.DO, containing pitch and duration values. We're ready to write a program that reads the file and plays the corresponding notes. Here it is:

```
100 REM--NAME:RDMUSC,BA-----
110 REM--Prgm reads file of pitch and
120 REM--duration values & plays notes
130 CLS
140 PRINT "The filename requested below"
150 PRINT "should contain pitch and"
160 PRINT "duration values of notes,"
170 PRINT "suffix .DO must be included"
180 INPUT "Filename"; FN$
185 '
190 OPEN FN$ FOR INPUT AS 1          ← Opens selected file to read file
195 '
200 '--beginning of GOTO loop-----
210   INPUT #1, P, DU
220   SOUND P, DU
230 IF NOT EOF(1) THEN 200
240 CLOSE
250 END
```

The new feature in this program is that we've used a *variable*, FS\$, for the name of the data file. This allows the user to enter *any* filename; that is, we can select one of several document files containing musical notes. Perfect! One program can select and play any composition that's properly stored in a document file! Using data files to store musical information offers much more flexibility and convenience than is possible with the DATA statements used in Chapter 16.

Sending Files to Different Locations

So far we've sent information to a RAM file from a BASIC program (writing a file) and from a RAM file to a BASIC program (reading a file). But we can use the OPEN statement to open files to devices other than random access memory (RAM). An example already mentioned is the cassette file, the name of which must have the prefix CAS:. Both RAM and CAS files are used for storing information. Figure 17-4 shows the names of other data files possible with the Model 100 and suggests the general way that information flows between these files and the BASIC program that manages them. In the following discussion, we briefly introduce each of these new types of file.

Two new kinds of files are called MDM ("MoDeM") and COM ("COMMunication"). These are communication files that generally transmit information between the Model 100 and another computer. A *modem file* is designed specifically for telephone communications using a modem as an intermediary device. A *communication file* transmits information through a standard

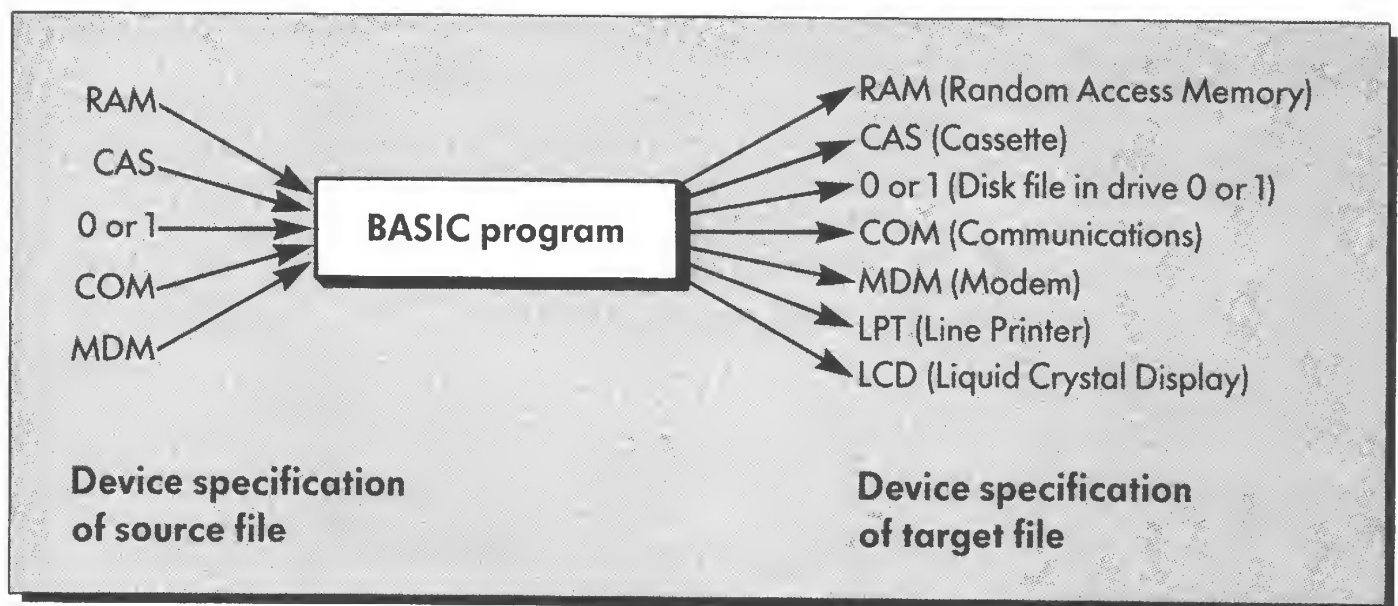


Figure 17-4. Possible information flow between different kinds of data files and a BASIC program

port, or outlet, called an RS 232 port. Though opening these files and sending and receiving information by means of the PRINT # and INPUT # statements is very similar to what we've already done with RAM files, the communication programs in which they are embedded are usually fairly complicated and very specific to the particular target computer. Because of the advanced and specialized nature of this use of files, we won't say more about them in this book. (If you do want to pursue this topic, you'll find the example program in the "Sample Session #2" of your *TRS-80® Model 100 Portable Computer* manual useful.)

We can also open a file to the printer, as well as to the screen. The complete filenames for these files are "LPT:" and "LCD:", respectively. Information can only be sent *to* these devices; that is, this type of file can only be opened for "OUTPUT".

Figure 17-4 suggests that a BASIC program can send a file from any one of the source files to any of the destination files. The following example shows how a BASIC program can read a RAM file containing pitch and duration values — we'll use the file BTHN9, which already exists in RAM — and send its contents to either the printer or the screen:

```
100 REM--NAME:TRAFIC,BA-----
110 REM--sends RAM file of pitch &
115 REM--duration values to printer or
120 REM--to screen,
130 CLS
135 MAXFILES = 2
138 '
140 INPUT "RAM file to send (ex: BTHN9.DO)"; F1$
150 INPUT "Destination (LPT: or LCD:)"; F2$
155 '
160 OPEN F1$ FOR INPUT AS 1
170 OPEN F2$ FOR OUTPUT AS 2
175 '
180 '--beginning of GOTO loop-----
190   INPUT #1, P, DU
200   PRINT #2, P, DU
210 IF NOT EOF(1) THEN 180
220 CLOSE
230 END
```

Notice that this program opens both the source and destination files at the same time but under different file numbers. This program can serve as a model for sending a file from any location to any destination.

Summary

In this chapter we have introduced one of the more difficult topics of BASIC programming — the data file or, more specifically, the sequential data file. A data file is a body of information that can be sent or received by a BASIC program. In most of this chapter, we have dealt with RAM files, which are used principally for storing data, such as pitch and duration values of musical notes, names and addresses, expenditures on a trip, and so forth. We can use CAS (cassette) files and disk files in a manner similar to the way we use RAM files. Other types of files, the MDM (modem) and COM (communication) files, involve communication between the MODEL 100 and other computers. LCD and LPT files send data to the screen and printer, respectively.

The following statements summarize how to set up and read sequential data files:

To set up or write a file:

1. “OPEN” the file “FOR OUTPUT” or “FOR APPEND”.
2. Write data to the file using PRINT #.
3. “CLOSE” a file when done writing to it.

To read a file:

1. “OPEN” the file “FOR INPUT”.
2. Read data from the file using INPUT #.
3. “CLOSE” the file when done.

It's Time to Say Goodbye

Well, we've come to the end of our story. We hope that you've enjoyed reading this book as much as the author has enjoyed writing it, and that you've learned even more than you'd hoped about BASIC. We have covered a lot of ground. We've introduced most of the BASIC instructions available on your Model 100, and we've explained many program techniques that enable you to assemble these instructions into efficient, powerful, and coherent BASIC programs. In short, you have all the tools necessary to write first-class BASIC programs. Good luck in your future programming endeavors on your Model 100!

Exercises

1. Write a program that reads a RAM data file "EXP1.DO", which contains an expense account in the following form: a string defining the nature of the expense and a numerical dollar amount. The file itself can be written with TEXT; show an example of what the file looks like.
2. Explain how you'd change the program written for exercise #1 to read a *disk* file EXP1.DO instead of a RAM file.

Solutions

1. The following shows the possible contents of the data file "RAM:EXP1.DO", as displayed and created with TEXT:

```
Hotel Tahiti Surf,      356.84 ◀
can of sardines,       2.86 ◀
beach ball,           4.55 ◀
gum ball,              .50 ◀
computer tape cassette, 5.45 ◀ ← Pressing ENTER here is optional
```

The spacing is not important, but string values must end with a comma. In place of using commas, quotation marks can be used around the string values. The following program will read, print each value, and find the total of all expenses:

```
100 REM--NAME:EXPACT,BA-----
110 REM--Prgm reads "RAM:EXP1.DO" &
120 REM--Prints entries and sum
130 CLS
135 SUM = 0
140 OPEN "RAM:EXP1.DO" FOR INPUT AS 1
150 '--beginning of GOTO loop-----
160 INPUT #1, ITEM$, CST
170 PRINT ITEM$, CST
180 SUM = SUM + CST
190 IF NOT EOF(1) THEN 150
200 CLOSE
205 PRINT
210 PRINT "Total expenses =" ; SUM
220 END
```

2. To read the disk file EXP1.DO instead of the RAM file EXP1.DO, change line 140 to read

```
140 OPEN "0:EXP1.DO" FOR INPUT AS 1
```

The assumption here is that EXP1.DO had previously been saved *with* the extension DO; if the extension was omitted in creating the document file, then the extension should be omitted in the OPEN statement.



Reserved Words

The following list of Reserved Words makes up the BASIC vocabulary on the Model 100. These words cannot be used as *any part* of a variable name.

AND	END	LOAD
AS	EOF	LOADM
ASC	EQV	LOG
ATN	ERL	LPOS
BEEP	ERR	LPRINT
CALL	ERROR	MAXFILES
CDBL	EXP	MAXRAM
CHR	FILES	MDM
CINT	FIX	MENU
CLEAR	FOR	MERGE
CLOAD	FRE	MID\$
CLOADM	GOSUB	MOTOR
CLOSE	GOTO	NAME
CLS	HIMEM	NEW
COM	IF	NEXT
CONT	IMP	NOT
COS	INKEY\$	OFF
CSAVE	INP	ON
CSAVEM	INPUT	OPEN
CSNG	INSTR	OR
CSRLIN	INT	OUT
DATA	IPL	PEEK
DATE\$	KEY	POKE
DAY	KILL	POS
DEFDBL	LCOPY	POWER
DEFINT	LEFT\$	PRESET
DEFSNG	LEN	PRINT
DEFSTR	LET	PSET
DIM	LINE	READ
EDIT	LIST	REM
ELSE	LLIST	RESTORE

RESUME
RIGHT\$
RND
RUN
RUNM
SAVE
SAVEM
SCREEN
SGN

SIN
SOUND
SPACE\$
SQR
STEP
STOP
STR\$
STRING
TAB

TAN
THEN
TIME\$
TO
USING
VAL
VARPTR
WIDTH
XOR

The following are additional Reserved Words if *disk*-BASIC is used:

DSKI\$
DSKO\$

LFILES
LOC

LOF















ASCII Character Codes

Decimal Value	Printed Character	Keyboard Character
0		PAUSE
1		CTRL A
2		CTRL B
3		CTRL C
4		CTRL D
5		CTRL E
6		CTRL F
7		CTRL G
8		CTRL H
9		CTRL I
10		CTRL J
11		CTRL K
12		CTRL L
13		CTRL M
14		CTRL N
15		CTRL O
16		CTRL P
17		CTRL Q
18		CTRL R
19		CTRL S
20		CTRL T
21		CTRL U
22		CTRL V
23		CTRL W
24		CTRL X
25		CTRL Y
26		CTRL Z
27		ESC
28		→
29		←
30		↑
31		↓

Decimal Value	Printed Character	Keyboard Character
32		SPACEBAR
33	!	!
34	"	"
35	#	#
36	\$	\$
37	%	%
38	&	&
39	,	,
40	((
41))
42	*	*
43	+	+
44	,	,
45	—	—
46	.	.
47	/	/
48	0	0
49	1	1
50	2	2
51	3	3
52	4	4
53	5	5
54	6	6
55	7	7
56	8	8
57	9	9
58	:	:
59	;	;
60	<	<
61	=	=
62	>	>
63	?	?
64	@	@
65	A	A
66	B	B
67	C	C
68	D	D
69	E	E
70	F	F

Decimal Value	Printed Character	Keyboard Character
71	G	G
72	H	H
73	I	I
74	J	J
75	K	K
76	L	L
77	M	M
78	N	N
79	O	O
80	P	P
81	Q	Q
82	R	R
83	S	S
84	T	T
85	U	U
86	V	V
87	W	W
88	X	X
89	Y	Y
90	Z	Z
91	[[
92	\	GRPH—
93]]
94	^	^
95	—	—
96	\	GRPH[
97	a	A
98	b	B
99	c	C
100	d	D
101	e	E
102	f	F
103	g	G
104	h	H
105	i	I
106	j	J
107	k	K
108	l	L
109	m	M

Decimal Value	Printed Character	Keyboard Character
110	n	N
111	o	O
112	p	P
113	q	Q
114	r	R
115	s	S
116	t	T
117	u	U
118	v	V
119	w	W
120	x	X
121	y	Y
122	z	Z
123	{	GRPH 9
124		GRPH —
125	}	GRPH 0
126	~	GRPH]
127		DEL
128		GRPH p
129		GRPH m
130		GRPH f
131		GRPH x
132		GRPH c
133		GRPH a
134		GRPH h
135		GRPH t
136	i	GRPH l
137	✓	GRPH r
138	≠	GRPH /
139	Σ	GRPH s
140	≈	GRPH '
141	±	GRPH =
142	∫	GRPH i
143	◀	GRPH e
144		GRPH y
145		GRPH u
146	↕	GRPH ;
147		GRPH q
148		GRPH w

Decimal Value	Printed Character	Keyboard Character
149	♂	(GRPH) b
150	♀	(GRPH) n
151	%	(GRPH) .
152	↑	(GRPH) o
153	↓	(GRPH) ,
154	→	(GRPH) l
155	←	(GRPH) k
156	♣	(GRPH) 2
157	♦	(GRPH) 3
158	♥	(GRPH) 4
159	♠	(GRPH) 5
160	'	(CODE) '
161	à	(CODE) x
162	ç	(CODE) c
163	£	(GRPH) 8
164	`	(CODE) "
165	μ	(CODE) M
166		(CODE))
167	▼	(CODE) —
168	†	(CODE) +
169	§	(CODE) s
170	¶	(CODE) R
171	©	(CODE) C
172	¼	(CODE) p
173	¾	(CODE) ;
174	½	(CODE) /
175	¶	(CODE) 0
176	¥	(GRPH) 7
177	Ä	(CODE) A
178	Ö	(CODE) O
179	Ü	(CODE) U
180	¢	(GRPH) 6
181	-	(CODE) [
182	ä	(CODE) a
183	ö	(CODE) o
184	ü	(CODE) u
185	ß	(CODE) S
186	™	(CODE) T
187	é	(CODE) d

Decimal Value	Printed Character	Keyboard Character
188	ù	CODE ,
189	è	CODE v
190	..	CODE =
191	f	CODE F
192	â	CODE l
193	ê	CODE 3
194	î	CODE 8
195	ô	CODE 9
196	û	CODE 7
197	^	CODE -
198	ë	CODE e
199	ï	CODE i
200	á	CODE q
201	í	CODE k
202	ó	CODE l
203	ú	CODE j
204	ý	CODE y
205	ñ	CODE n
206	ã	CODE z
207	õ	CODE .
208	Â	CODE !
209	Ê	CODE #
210	Î	CODE *
211	Ô	CODE (
212	Û	CODE &
213	Ï	CODE I
214	Ë	CODE E
215	É	CODE D
216	Á	CODE Q
217	Í	CODE K
218	Ó	CODE L
219	Ú	CODE J
220	Ý	CODE Y
221	Ù	CODE <
222	È	CODE V
223	À	CODE X
224		GRPH Z
225	▪ (upper left)	GRPH !
226	▪ (upper right)	GRPH @

Decimal Value	Printed Character	Keyboard Character
227	■ (lower left)	GRPH #
228	■ (lower right)	GRPH \$
229	▤	GRPH %
230	▥	GRPH ^
231	▧ (upper)	GRPH Q
232	▨ (lower)	GRPH W
233	▩ (left)	GRPH E
234	▪ (right)	GRPH R
235	▫	GRPH A
236	▬	GRPH S
237	▭	GRPH D
238	▮	GRPH F
239	▯	GRPH X
240	▰	GRPH U
241	▱	GRPH P
242	▲	GRPH O
243	△	GRPH I
244	▴	GRPH J
245	▵	GRPH :
246	▶	GRPH M
247	▷	GRPH >
248	▸	GRPH <
249	▹	GRPH L
250	►	GRPH K
251	▻	GRPH H
252	▼	GRPH T
253	▽	GRPH G
254	▾	GRPH Y
255	▿	GRPH C



BASIC Error Messages

When BASIC encounters an error, it prints out an error message containing a two-letter code that identifies the type of error. The following table lists these error codes and their meaning.

Message	Meaning
AO	Already open
AT	Bad allocation table*
BN	Bad file number
BS	Bad subscript
CF	File not open
CN	Can't continue
DD	Doubly dimensioned array
DF	Disk full*
DN	Bad drive number*
DS	Direct statement in file
EF	Input past end of file
FC	Illegal function call
FE	File already exists*
FF	File not found
FL	Undefined error
ID	Illegal direct
IE	Undefined error
IO	Error
LS	String too long
MO	Missing operand
NF	NEXT without FOR
NM	Bad file name
NR	No RESUME
OD	Out of data
OM	Out of memory
OS	Out of string space
OV	Overflow

*Used only in *disk*-BASIC

RG	RETURN without GOSUB
RW	RESUME without error
SN	Syntax error
ST	String formula too complex
TM	Type mismatch
TS	Bad track/sector*
UE	Undefined error
UL	Undefined line
/O	Division by zero

Index

- !, 161, 216
- ! tag, 153
- #, 161, 216
 - place holders, 163
 - symbol, 163
 - tag, 155
- \$, 100, 216
- %, 161, 216
- % tag, 159

- A used in SAVE command, 76
- ABS function, 246, 249
- ADDRSS, 3, 4
- AND operator, 144
- ASC, 282
- ASCII, 76, 179-80
 - character, 183
 - character codes, 272
 - character set, 182
 - code, 41, 173, 272-73
 - defined, 145
- ATN function, 255
- Active memory, 65, 86
- Adding strings, 102
- Addition, 24-5, 99, 106, 245
- Adjacent side, 254
- All-points addressable graphics, 172, 226
- Angle, 253
- Apostrophe, 56
- Apostrophe for REM statement, 56
- Append, 49
- Arccos, 257
- Arcsin, 257
- Area of a circle, 258
- Arithmetic, 24
 - operations, 245
 - with variables, 97
- Array(s), 206, 213-15, 219, 221
- Array element(s), 214, 216, 221
- Arrow in EDIT mode, 45
- Assign, 135, 137, 143
- Assigning variables, 105
 - numeric variables, 93
 - string variables, 101

- Asterisk(s), 21, 25, 166-67
- Asterisks (*) in PRINT USING, 166

- BA extension, 67, 85, 87, 307, 317
- BASIC, 1, 2, 4, 7
 - command(s), 12, 16
 - output, 19
 - program, 11
 - statement(s), 12, 13, 16
- BEEP instruction, 10, 17
- BREAK function, 44
- BREAK key, 44
- Bar graph, 202-3
- Beethoven's Ninth Symphony, 295, 298, 318
- Blinking cursor, 113-14, 274
- Body of the loop, 129
- Box option with LINE, 238
- Boxed title, 177
- Boxes, 234
- Branch, 126
- Branching, 125
- Bytes, 159

- CAPS LOCK key, 8
- CAS (for "cassette"), 80
- CAS device specification, 307
- CAS file(s), 81, 320
- CHAR\$, 282
- CHR\$ function, 42, 180-83
- CINT function, 247
- CLOAD command, 82, 84, 88
- CLOSE statement, 310-11, 314
- CLS command, 10, 17, 227
- CODE key, 173, 178-79, 180, 183
- COM file(s), 320, 322
- CONT command, 44, 306
- COS function, 251, 254
- CSAVE command, 81, 88, 304
- CTRL BREAK, 197, 199
- CTRL key, 46, 175
- Calling a subroutine, 185

- Caret symbols (^) in PRINT USING, 168
- Carriage return, 23, 35, 180
- Cassette file(s), 79, 80, 88, 307
- Cassette recorder, 3, 79
- Chaining programs, 71
- Character(s), 21, 95, 172
 - coordinate, 32-34, 109
 - graphics, 172, 226
 - positions, 226
- Clause, 141, 146
- Clusters, 86
- Colon, 144, 307
- Column(s), 28, 30-31, 109
 - number, 225
 - position, 175
- Comma(s), 118, 120-21, 166
- Commas (,) in PRINT USING, 166
- Command keys, 40, 43
- Communication files, 320
- Compounded, 134
- Concatenation, 102-3
- Controlled loops, 129
- Coordinates, 27, 235
- Copying BASIC programs, 75
- Corners, 34
- Correcting, 13
- Cosine, 251
- Counter variable name, 129, 132
- Current program, 65
- Cursor, 7, 42, 117
- Cursor control key(s), 40, 46, 50
- Cycloid, 265

- DATA, 207
 - list, 212
 - statement(s), 206-7, 209-10, 214-16, 295-96, 298, 305
 - values, 208
- DATE\$, 42
- DD error, 217
- DEFDBL statement, 161
- DEFINT statement, 161
- DEFSNG statement, 161
- DEFSTR statement, 161

- DEL/BKSP key, 9, 44, 47, 61, 319
- DIM statement, 216-19, 223
- DO extension, 307, 317
- Dash, 21
- Data file(s), 69, 304-5, 312, 318, 322
- Decimal point, 165, 168
- Decimal points (.) in PRINT USING, 165
- Decimal value, 151
- Decisions, 140
- Decision making, 125
- Declaration statement, 162
- Degrees, 252-53, 256
- Deleting, 13
- Deleting characters, 47
- Destination files, 321
- Device specification, 69, 80, 86, 319
- Diagonal elements, 225
- Digits, 151-52, 154, 270
- Dimension(s), 213
 - declaration, 217
 - of arrays, 218
- Direct mode, 9-11
- Disk, 85
 - drive numbers, 86
 - drive/video interface, 85
 - file(s), 85-8
- Dividing lines, 49
- Division, 24-5, 99, 106, 245
- Dollar sign (\$), 100-1, 115, 167, 216
- Dot graphics, 172, 226
- Dots, 173
- Double precision, 25
- Double-precision integer, 156
- Double-precision numbers, 151-52, 155, 159, 169
- Double-precision value, 154
- Duration, 287-89
- Duration value(s), 288, 296, 300, 318

- EDIT, 39, 43, 51, 62, 319
- EDIT mode, 44-45, 50
- EF error, 314
- ELSE option, 141

- END statement, 15, 17, 127, 144, 195
- ENTER key, 4, 8
- EOF statement, 314-16
- ERASE statement, 222-23
- EXP function, 259-60
- Edit, 45
- Editing long programs, 50
- End of file, 315
- Endless loop, 127-28
- English, 1
- Equals, 143
- Equal sign, 135, 137
- Equal to or greater than, 143
- Equal to or less than, 143
- Erasing a RAM file, 78
- Erasing a disk file, 87
- Erasing with LINE, 236
- Error message, 21, 74, 101
- Evaluating arithmetic expressions, 107
- Even numbers, 236
- Exclamation mark (!), 153-54
- Exclamation point (!) tag, 153
- Execute, 12
- Exponent, 158, 258
- Exponential notation, 168
- Exponential notation in PRINT USING, 167
- Expression, 141
- Extended ASCII, 180
- Extended character set, 173, 179
- Extra ignored, 119

- F1 key, 40, 197-98
- F1 for FILES, 77
- F2 key, 197, 198
- F2 for LOAD, 40, 77
- F3 for SAVE, 41, 77
- F5 for LIST, 40
- F6 key, 41, 62
- F7 key, 41, 62
- F8 key, 40, 66
- F8 for MENU, 41
- FC error, 74, 288
- FILES command, 72-73, 77, 79
- FIX function, 247-249
- FOR INPUT, 307
- FOR OUTPUT, 307
- FOR statement, 129-30, 132

- FOR...NEXT loop, 133, 135, 160, 178, 190, 202, 212, 215-16, 221, 229-30, 235, 237, 268, 282
- FOR...NEXT statement, 129, 132, 138, 147, 289
- FOR...NEXT..STEP loop, 129
- Fields, 316
- File(s), 3, 306
 - extension, 80, 85
 - list, 82
 - specification, 69
- Filename(s), 66, 68-69, 70, 74, 85
 - extension of, 67
- Filling boxes with LINE, 240
- Floating-point number, 169
- Floppy disk, 85
- Fold over, 59
- Folded statements, 59
- Foreign language letters, 173
- Format string(s), 163-65
- French, 178
- Frequency, 290-92, 302
- Function key(s), 39, 197

- GOSUB statement, 186-89, 202
- GOTO loop, 147
- GOTO statement, 126, 144, 147, 186, 188, 237
- GRPH ke, 173-80, 183
- German, 178
- Graph(s), 133, 139, 172, 227, 231
- Graphics, 172, 241, 254
 - images, 173
 - symbols, 269
- Greater than, 143

- Hard-copy, 56
- Horizontal position, 227
- House, 176
- Humuhumunukunukuapuaa, 284

- IF statement, 127
- IF...THEN statement, 140, 230
- IF...THEN...ELSE statement, 141-42, 147

- INKEY\$ function, 273, 275-77, 285
- INPUT, 112
 - error message, 119
 - statement, 112-14, 118, 122, 273
 - with prompt, 117
- INPUT # statement, 311-12, 316, 321
- INPUT\$ function, 274-75, 277, 285
- INSTR function, 283-85
- INT function, 247-249
- Illegal variable names, 96
- Immediate line, 10
- Increment, 135
- Index, 193
 - limits, 132
 - variable, 129-30, 193
- Indexed branching, 191
- Initializes, 137
- Inner loop, 139
- Input, 112
- Inputting numeric variables, 113
- Inputting string variables, 115
- Inserting characters, 46
- Inserting lines, 49
- Insertion, 46
- Integer(s), 136, 151, 159, 169, 216
 - arithmetic, 159
 - index variable, 160
 - numbers, 161
 - type, 159
 - variables, 161
- Interest, 134, 160
- Interrupting INPUT with SHIFT BREAK, 116
- Interrupting program execution, 128
- Inverse cosine, 257
- Inverse sine, 257
- Inverse tangent, 256
- Investment, 134
- Investment period, 149
- Joining lines, 49
- KEY command, 42
- KEY() OFF statement, 199-200
- KEY() ON statement, 196, 198, 202
- KEY() STOP statement, 199-200
- KILL command, 78-79, 87
- Keyboard characters, 271
- Kilobytes, 2, 86
- LABEL key, 40, 43, 77, 319
- LCOPY statement, 57, 58, 60-61
- LEFT\$ function, 279-80, 283, 285
- LEN functions, 278
- LFILES command, 86-87
- LINE, 235, 239
 - INPUT, 121, 123
 - INPUT statement, 121-22
 - statement, 234, 237-38, 241, 255, 263
 - with the B and BF options, 240
- LIST command, 14-15, 17, 23, 52-53
- LLIST command, 58-61, 110
- LOAD command, 70-74, 77, 79, 82, 84, 87-88
- LOG function, 259-60
- LPRINT statement, 32, 60-61, 135
- LPRINT USING, 167
- Leaderless magnetic tape, 80
- Leading blanks, 166
- Left-justified, 29
- Left-justified columns, 164
- Legal variable names, 96
- Less than, 143
- Line number(s), 13, 14, 147
- Linear program structures, 125
- Linking programs, 71
- Loading RAM files, 70
- Loading a cassette file, 81
- Logarithm, 260
- Logical BASIC line, 20
- Logical operator, 144
- Logs, 257
- Loop, 127-28, 147
- Loop index, 190, 233
- Looping, 125
- Lowercase letters, 94
- MAIN PROGRAM, 110
- MDM file, 320, 322
- MID\$ function, 281-283, 285
- MOD function, 183
- MUSIC, 287
- Main menu, 72
- Main menu cursor, 72
- Main program, 185
- Mathematical notation, 156
- Mathematical symbols, 173
- Maximum length, 119
- Maximum length of a BASIC line, 21
- Maximum line length, 59
- Maximum string length, 21
- Memory, 2
 - Memory space, 86, 159
 - Memory space bytes, 154
- Menu cursor, 4
- Model 100, 1
- Modem files, 320
- Modifiers, 53
- Monthly balance, 160
- Moving a character, 178
- Multiple DATA statements, 209
- Multiple READ statements, 210
- Multiple variables, 118
- Multiplication, 24-25, 99, 106, 245
- Music, 291, 300
- Musical staff, 292
- NAME, 79
- NAME...AS command, 78
- NEW command, 16-17, 67
- NEXT statement, 129-30, 132
- NUM key, 7
- Nameless load commands, 84
- Naming files, 67
- Naming variables, 94
- Natural logarithm, 260
- Negative exponent, 258
- Negative operands, 249
- Negative sign, 25
- Nested FOR...NEXT loops, 138, 225
- Nested loop(s), 139, 149
- Not equal to, 143
- Note value, 297
- Null string, 35, 277
- Number(s), 24, 28, 150

- symbols (#) in PRINT USING, 164
- types, 150
- Numeric constant, 115
- Numeric expression(s), 26, 103-4, 268-69
- Numeric function(s), 245-46, 267
- Numeric value(s), 142, 270
- Numeric variable, 91, 208, 267

- OD error, 208, 212
- ON GOSUB...RETURN statement, 196
- ON...GOSUB statement, 191, 193, 195, 202, 275
- ON...KEY GOSUB statement, 186, 196-198, 200, 202
- OPEN statement, 307, 320
- OPEN...FOR APPEND statement, 313-14
- OPEN...FOR INPUT, 316
- OPEN...FOR OUTPUT, 314
- OR operator, 144
- OV error, 248
- Octaves, 302
- Octave value, 297
- Odd number, 236
- Ok, 6
- One-dimensional array, 220
- Operand(s), 252, 268, 272, 281
- Opposite side, 254
- Order of Operations, 108
- Outer loop, 139
- Output, 112
- Outside loop, 140

- PASTE key, 43
- PAUSE function, 44
- PAUSE key, 43, 61, 128, 135
- PAUSE/BREAK key, 43, 61, 128
- PRESET statement, 233-34, 241
- PRINT key, 17, 24, 43, 57, 60-61, 91
- PRINT statement, 19, 22-23, 26, 35-36, 102
- PRINT with semicolons, 28
- PRINT with spaces, 28
- PRINT # statement, 309, 311, 316, 321
- PRINT @ statement, 19, 31-2, 35-7, 109, 175, 182-83
- PRINT USING statement, 150, 162, 164-65, 169
- PSET statement, 23, 227-229, 232, 241, 255
- Parabola, 231
- Parentheses, 106-8, 214, 228, 246, 274
- Percent sign (%), 161, 164
- Percent tag, 159
- Physical line, 20
- Pi, 259
- Pitch value, 288, 292, 296, 300, 302
- Pitch, 287-89, 318
- Pixel(s), 172-73, 226, 228, 233-34
- Place holders, 168
- Play button, 82
- Play mode, 82
- Pointer, 75
- Positive numbers, 25
- Powers, 257
- Power of e, 259
- Precision, 152
- Present value, 134
- Print, 9, 10
- Printer, 3, 39, 56, 167
- Printing numeric variable expressions, 99
- Printing with TAB, 31-32
- Printing with commas, 29-30
- Printing with semicolons, 26
- Printing with spaces, 26
- Program line(s), 12, 16
- Program listings, 58-59
- Program mode, 11
- Program organization, 185
- Programmable function keys, 40, 76
- Prompt, 6
- Prompt with INPUT, 117
- Pseudo-random number generator, 262

- Question mark(s), 35, 42, 113-14, 117-18, 120-21, 209, 274

- R used in LOAD command, 84
- RAM (Random Access Memory), 64, 66, 320
- RAM file(s), 64-67, 70, 72-75, 87-88, 160, 199, 321
- READ statement(s), 206-7, 212, 209-10, 214-16
- REM statement(s), 54-5, 62, 110
- RESTORE statement, 212, 213
- RETURN statement, 186, 188-89, 193, 195, 198, 202
- RIGHT\$ function, 277, 280, 283, 285
- RND function, 261, 262-63, 265
- ROM (Read Only Memory), 66
- RUN command, 12, 13, 17, 52, 54
- Radians, 256
- Radian measure, 252
- Radius, 258
- Random access memory, 64
- Random numbers, 261-62
- Read only memory, 66
- Reading data files, 304
- Reciprocal, 258
- Records, 316
- Record button, 80
- Redimension n array, 223
- Redo from start, 119
- Relational operators, 142, 143
- Remarks, 54
- Renaming a RAM file, 77
- Replacing characters, 47
- Reserved word(s), 8, 95, 105, 115
- Resolution, 226
- Right-justified columns, 164
- Roots, 257
- Row, 33, 109, 225
- Row position, 175
- Ruler, 31
- Rules for naming variables, 94
- Rules for numeric variable names, 97
- Rules of order, 106

- SAVE command, 66-67, 69, 71, 76, 77, 79-80, 86, 88
- SCHEDL, 3-4
- SGN function, 247

- SHIFT key, 15, 43, 46-47, 174-75
- SHIFT BREAK keys, 128
- SHIFT DEL/BKSP keys, 49
- SIN function, 251, 254, 265
- SN error, 8, 115
- SOUND statement, 287-92, 297, 300
- SPACE\$ function, 268-69, 285
- SQR function, 259
- SQUARE function, 134, 139-40, 231-32
- STEP option, 135-36, 138
- STR\$ function, 269, 270-71, 285
- STRING\$ function, 271-73, 285
- Saving RAM files, 66
- Saving a cassette file, 80
- Saving a file in RAM, 66
- Scientific notation, 155-56, 158, 169
- Screen coordinate, 175, 182, 205
- Screen width, 34
- Scroll, 176
- Scrolled, 35
- Scrolling, 135
- Semicolon, 132, 163, 317
- Semicolon after even, 238
- Sequence of numbers, 26
- Sequential data file, 312, 322
- Significant digits, 168
- Sine, 251
- Single-precision numbers, 152-53, 155, 159, 169
- Single-precision variable, 154
- Single-precision value, 154
- Slash, 25
- Software, 5
- Source files, 321
- Special characters, 173
- Special symbols, 173
- Spring value, 268
- Square, 231
- Square root, 259
- Standard notation, 156
- Stopwatch, 277
- Storing programs, 64
- String(s), 20, 28, 35, 100, 278
 - array, 217
 - constant(s), 100-1, 115, 142, 164
 - function(s), 41, 181, 267, 298
 - value(s), 142, 167, 270
 - variable(s), 99, 101, 103, 142, 162, 164, 208-9
- Subroutine(s), 185, 187, 189-91, 205
- Subscript(s), 213-17
- Subtraction, 24-25, 99, 106, 245
- Switches, 1
- Switch value, 238
- Switch variable, 237
- Syntax error, 8
- TAB instruction, 19, 31
- TAB key, 133, 175
- TAN function, 251, 252, 254
- TELCOM, 4
- TEXT, 4, 318
 - how to use, 318
 - program, 44
 - used to write DATA files, 317
- TIME\$ function, 42, 277
- TIME\$ variable, 286
- TM error, 101
- TRS-80 Model 100, 2
- Tangent, 251-52
- Text, 3
- Text lines, 49
- Three-dimensional array, 219
- Tone, 288
- Triangle in EDIT mode, 45
- Trigonometric functions, 251, 254, 257
- Trigonometry, 251
- Truncates, 249
- Two-dimensional array, 218, 221
- Type declaration statement, 169
- Type declaration tag(s), 154, 161, 169, 216
- Type mismatch, 101, 119
- Type tag, 159
- Underline, 21
- Uppercase letters, 94
- Uppercase, 174
- VAL function, 269-71, 285
- Value of a variable, 92
- Variable(s), 91, 98, 101, 114, 122, 132, 154, 206, 213
 - boxes, 93
 - declaration statements, 161
 - names, 213
 - type, 115, 208
- Vertical formatting, 22
- Very large numbers, 155
- Very small numbers, 155
- Words, 3
- Writing data files, 304
- Writing to a file, 309
- X-coordinate, 227-28, 232-35, 240, 255
- XOR operator, 144
- Y-coordinate, 227-28, 232, 235, 240, 255
- Zero, 236
- Zeroth element, 217



(0452)

Other PLUME/WAITE books on the TRS-80® Model 100:

- ☐ **Introducing the TRS-80® Model 100, by Diane Burns and S. Venit.** This book, intended for newcomers to the Model 100, offers simple step-by-step explanations of how to set up your Model 100 and how to use its built-in programs: TEXT, ADDRSS, SCHEDL, TELCOM, and BASIC. Specific instructions are given for connecting the Model 100 to the cassette recorder, other computers, the telephone lines, the optional disk drive/video interface, and the optional bar code reader. (255740—\$15.95)
- ☐ **Practical Finance on the TRS-80® Model 100, by S. Venit and Diane Burns.** The perfect book for anyone using the Model 100 in business: investors, real estate brokers, managers. Contains short but powerful programs to perform production planning, and access financial and other information from CompuServe® and the Dow Jones News/Retrieval® service. (255767—\$15.95)
- ☐ **Games and Utilities for the TRS-80® Model 100, by Ron Karr, Steven Olsen, and Robert Lafore.** A collection of powerful programs to enhance your Model 100. Enjoy fast-paced, exciting card games, arcade games, music, art, and learning games. Help yourself to practical utilities that let you count words in a text file, turn your Model 100 into a scientific calculator, show file sizes, and generally increase your Model 100's usefulness, and your own grasp of programming. (255775—\$16.95)
- ☐ **Hidden Powers of the TRS-80® Model 100, by Christopher L. Morgan.** This amazing book takes you deep inside the Model 100 to reveal for the first time how it really works. You'll learn about the amazing power buried in the ROM, and how to use this power in your own programs. You can print in reverse video, prevent any screen lines from scrolling, dial the telephone from BASIC, control external devices from the cassette port, and discover many other fascinating secrets hidden within your Model 100. (255783—\$19.95)

All prices higher in Canada.

To order, use the convenient coupon on the next page.



Other PLUME/WAITE books available

(0452)

- ☐ **BASIC PRIMER for the IBM® PC and XT by Bernd Enders and Bob Petersen.** An exceptionally easy-to-follow entry into BASIC programming that also serves as a comprehensive reference guide for the advanced user. Includes thorough coverage of all IBM BASIC features: color graphics, sound, disk access, and floating point.
(254957 — \$16.95)
- ☐ **DOS PRIMER for the IBM® PC and XT by Mitchell Waite, John Angermeyer and Mark Noble.** An easy-to-understand guide to IBM's disk operating system, versions 1.1 and 2.0, which explains — from the ground up — what a DOS does and how to use it. Also covered are advanced topics such as the fixed disk, tree-structured directories, and redirection.
(254949 — \$14.95)
- ☐ **PASCAL PRIMER for the IBM® PC by Michael Pardee.** An authoritative guide to this important structured language. Using sound and graphics examples, this book takes the reader from simple concepts to advanced topics such as files, linked lists, compilands, pointers, and the heap.
(254965 — \$17.95)
- ☐ **ASSEMBLY LANGUAGE PRIMER for the IBM® PC and XT by Robert Lafore.** This unusual book teaches assembly language to the beginner. The author's unique approach, using DEBUG and DOS functions, gets the reader programming fast without the usual confusion and overhead found in most books on this fundamental subject. Covers sound, graphics, and disk access.
(254973 — \$21.75)
- ☐ **BLUEBOOK OF ASSEMBLY ROUTINES for the IBM® PC and XT by Christopher Morgan.** A collection of expertly written "cookbook" routines that can be plugged in and used in any BASIC, Pascal, or assembly language program. Included are graphics, sound, arithmetic conversions. Get the speed and power of assembly language in your program, even if you don't know the language!
(254981 — \$19.95)

All prices higher in Canada.

Buy them at your local bookstore or use this convenient
coupon for ordering.

NEW AMERICAN LIBRARY

P.O. Box 999, Bergenfield, New Jersey 07621

Please send me the PLUME BOOKS I have checked above. I am enclosing \$_____ (please add \$1.50 to this order to cover postage and handling). Send check or money order—no cash or C.O.D.'s. Prices and numbers are subject to change without notice.

Name_____

Address_____

City_____State_____Zip Code_____

Allow 4-6 weeks for delivery
This offer subject to withdrawal without notice.

The Waite Group

COMPUTER • Z5575 • \$19.95

CANADA • \$24.95

MASTERING BASIC ON THE TRS-80® MODEL 100

Here is the only book that allows complete mastery of the powerful BASIC language contained in the Model 100 portable computer. Using an exceptionally easy, at-the-keyboard approach that is packed with entertaining and practical examples, you'll learn how to control the liquid crystal display, the sound generator, and the Model 100's unique file-handling techniques. This is the first book on Model 100 BASIC to thoroughly explain programming the new optional disk drive. Also revealed are the tricks and techniques of veteran BASIC programmers, so you can learn to write your own efficient software.

Imaginative example programs, exciting games, summary boxes, extensive graphics and review exercises make this an indispensable primer and reference for every Model 100 owner.

The Waite Group is a Sausalito, California, based producer of high-quality books on personal computing. Acknowledged as a leader in the industry, the Waite Group has written and produced over thirty titles, including such best sellers as Assembly Language Primer for the IBM PC & XT, Graphics Primer for the IBM PC, CP/M Primer, and Soul of CP/M. Internationally known and award winning, Waite Group books are distributed worldwide, and have been repackaged with the products of such major companies as Epson, Wang, Xerox, Tandy Radio-Shack, NCR and Exxon. Mr. Waite, President of the Waite Group, has been involved in the computer industry since 1976 when he bought his first Apple I computer from Steven Jobs.

